O'REILLY®
オライリー・ジャパン

TURING 图灵程序设计丛书

深度学习进阶

自然语言处理

Natural Language Processing

[日] 斋藤康毅 著
陆宇杰 译


中国工信出版集团


人民邮电出版社
POSTS & TELECOM PRESS

作者介绍

斋藤康毅

1984年出生于日本长崎县，东京工业大学毕业，并完成东京大学研究生院课程。目前在某企业从事人工智能相关的研究和开发工作。著有《深度学习入门：基于Python的理论与实现》，同时也是*Introducing Python*、*Python in Practice*、*The Elements of Computing Systems*、*Building Machine Learning Systems with Python*的日文版译者。

译者介绍

陆宇杰

长期从事自然语言处理、知识图谱、深度学习相关的研究和开发工作。译有《深度学习入门：基于Python的理论与实现》。

数字版权声明



图灵程序设计丛书

深度学习进阶：自然语言处理

Natural Language Processing

[日] 斋藤康毅 著

陆宇杰 译

O'REILLY®

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'Reilly Japan, Inc. 授权人民邮电出版社出版

人民邮电出版社

北　　京

图书在版编目（CIP）数据

深度学习进阶 ：自然语言处理 /（日）斋藤康毅著 ；
陆宇杰译. -- 北京 ：人民邮电出版社，2020.10
（图灵程序设计丛书）
ISBN 978-7-115-54764-4

Ⅰ. ①深… Ⅱ. ①斋… ②陆… Ⅲ. ①自然语言处理
Ⅳ. ①TP391

中国版本图书馆CIP数据核字(2020)第168652号

内 容 提 要

本书是《深度学习入门：基于 Python 的理论与实现》的续作，围绕自然语言处理和时序数据处理，介绍深度学习中的重要技术，包括 word2vec、RNN、LSTM、GRU、seq2seq 和 Attention 等。本书语言平实，通俗易懂，结合大量示意图和 Python 代码，遵循“提出问题”“思考解决问题的新方法”“加以改善”的方法，基于深度学习解决自然语言处理相关的各种问题，使读者在此过程中更深入地理解深度学习中的重要技术。

本书适合对自然语言处理感兴趣的读者阅读。

◆ 著　　　　[日] 斋藤康毅
译　　　　陆宇杰
责任编辑　杜晓静
责任印制　周昇亮
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　https://www.ptpress.com.cn
北京　　　　　　　　印刷
◆ 开本：880×1230　1/32
印张：13.125　　　　彩插：1
字数：394千字　　　　2020年10月第1版
印数：1－3 500册　　　2020年10月北京第1次印刷
著作权合同登记号　图字：01-2018-7738号

定价：99.00元

读者服务热线：(010)51095183转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东市监广登字20170147号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

译者序 ······ xi

前言 ······ xiii

第1章　神经网络的复习 ······ 1

1.1　数学和Python的复习 ······ 1

1.1.1　向量和矩阵 ······ 1

1.1.2　矩阵的对应元素的运算 ······ 4

1.1.3　广播 ······ 4

1.1.4　向量内积和矩阵乘积 ······ 6

1.1.5　矩阵的形状检查 ······ 7

1.2　神经网络的推理 ······ 8

1.2.1　神经网络的推理的全貌图 ······ 8

1.2.2　层的类化及正向传播的实现 ······ 14

1.3　神经网络的学习 ······ 18

1.3.1　损失函数 ······ 18

1.3.2　导数和梯度 ······ 21

1.3.3　链式法则 ······ 23

1.3.4　计算图 ······ 24

1.3.5　梯度的推导和反向传播的实现 ······ 35

1.3.6　权重的更新 ······ 39

1.4　使用神经网络解决问题 ······ 41

1.4.1　螺旋状数据集 ······ 41

1.4.2　神经网络的实现 ······ 43

1.4.3　学习用的代码 ······ 45

1.4.4　Trainer 类 ······ 49

1.5　计算的高速化 ······ 50

1.5.1　位精度 ······ 51

1.5.2 GPU（CuPy） 52
1.6 小结 54

第2章 自然语言和单词的分布式表示 57
2.1 什么是自然语言处理 57
2.2 同义词词典 59
2.2.1 WordNet 61
2.2.2 同义词词典的问题 61
2.3 基于计数的方法 63
2.3.1 基于Python的语料库的预处理 63
2.3.2 单词的分布式表示 66
2.3.3 分布式假设 67
2.3.4 共现矩阵 68
2.3.5 向量间的相似度 72
2.3.6 相似单词的排序 74
2.4 基于计数的方法的改进 77
2.4.1 点互信息 77
2.4.2 降维 81
2.4.3 基于SVD的降维 84
2.4.4 PTB数据集 86
2.4.5 基于PTB数据集的评价 88
2.5 小结 91

第3章 word2vec 93
3.1 基于推理的方法和神经网络 93
3.1.1 基于计数的方法的问题 94
3.1.2 基于推理的方法的概要 95
3.1.3 神经网络中单词的处理方法 96
3.2 简单的word2vec 101
3.2.1 CBOW模型的推理 101
3.2.2 CBOW模型的学习 106
3.2.3 word2vec的权重和分布式表示 108
3.3 学习数据的准备 110
3.3.1 上下文和目标词 110
3.3.2 转化为one-hot表示 113

3.4 CBOW模型的实现 …… 114
3.5 word2vec的补充说明 …… 120
3.5.1 CBOW模型和概率 …… 121
3.5.2 skip-gram模型 …… 122
3.5.3 基于计数与基于推理 …… 125
3.6 小结 …… 127

第4章 word2vec的高速化 …… 129
4.1 word2vec的改进① …… 129
4.1.1 Embedding层 …… 132
4.1.2 Embedding层的实现 …… 133
4.2 word2vec的改进② …… 137
4.2.1 中间层之后的计算问题 …… 138
4.2.2 从多分类到二分类 …… 139
4.2.3 sigmoid函数和交叉熵误差 …… 141
4.2.4 多分类到二分类的实现 …… 144
4.2.5 负采样 …… 148
4.2.6 负采样的采样方法 …… 151
4.2.7 负采样的实现 …… 154
4.3 改进版word2vec的学习 …… 156
4.3.1 CBOW模型的实现 …… 156
4.3.2 CBOW模型的学习代码 …… 159
4.3.3 CBOW模型的评价 …… 161
4.4 wor2vec相关的其他话题 …… 165
4.4.1 word2vec的应用例 …… 166
4.4.2 单词向量的评价方法 …… 168
4.5 小结 …… 170

第5章 RNN …… 173
5.1 概率和语言模型 …… 173
5.1.1 概率视角下的word2vec …… 174
5.1.2 语言模型 …… 176
5.1.3 将CBOW模型用作语言模型？ …… 178
5.2 RNN …… 181
5.2.1 循环的神经网络 …… 181

5.2.2 展开循环 ······ 183
5.2.3 Backpropagation Through Time ······ 185
5.2.4 Truncated BPTT ······ 186
5.2.5 Truncated BPTT 的 mini-batch 学习 ······ 190
5.3 RNN 的实现 ······ 192
5.3.1 RNN 层的实现 ······ 193
5.3.2 Time RNN 层的实现 ······ 197
5.4 处理时序数据的层的实现 ······ 202
5.4.1 RNNLM 的全貌图 ······ 202
5.4.2 Time 层的实现 ······ 205
5.5 RNNLM 的学习和评价 ······ 207
5.5.1 RNNLM 的实现 ······ 207
5.5.2 语言模型的评价 ······ 211
5.5.3 RNNLM 的学习代码 ······ 213
5.5.4 RNNLM 的 Trainer 类 ······ 216
5.6 小结 ······ 217

第 6 章 Gated RNN ······ 219
6.1 RNN 的问题 ······ 220
6.1.1 RNN 的复习 ······ 220
6.1.2 梯度消失和梯度爆炸 ······ 221
6.1.3 梯度消失和梯度爆炸的原因 ······ 223
6.1.4 梯度爆炸的对策 ······ 228
6.2 梯度消失和 LSTM ······ 229
6.2.1 LSTM 的接口 ······ 230
6.2.2 LSTM 层的结构 ······ 231
6.2.3 输出门 ······ 234
6.2.4 遗忘门 ······ 236
6.2.5 新的记忆单元 ······ 237
6.2.6 输入门 ······ 238
6.2.7 LSTM 的梯度的流动 ······ 239
6.3 LSTM 的实现 ······ 240
6.4 使用 LSTM 的语言模型 ······ 248
6.5 进一步改进 RNNLM ······ 255
6.5.1 LSTM 层的多层化 ······ 256

6.5.2 基于Dropout抑制过拟合 …… 257
6.5.3 权重共享 …… 262
6.5.4 更好的RNNLM的实现 …… 263
6.5.5 前沿研究 …… 269
6.6 小结 …… 270

第7章 基于RNN生成文本 …… 273
7.1 使用语言模型生成文本 …… 274
7.1.1 使用RNN生成文本的步骤 …… 274
7.1.2 文本生成的实现 …… 278
7.1.3 更好的文本生成 …… 281
7.2 seq2seq模型 …… 283
7.2.1 seq2seq的原理 …… 283
7.2.2 时序数据转换的简单尝试 …… 287
7.2.3 可变长度的时序数据 …… 288
7.2.4 加法数据集 …… 290
7.3 seq2seq的实现 …… 291
7.3.1 Encoder类 …… 291
7.3.2 Decoder类 …… 295
7.3.3 Seq2seq类 …… 300
7.3.4 seq2seq的评价 …… 301
7.4 seq2seq的改进 …… 305
7.4.1 反转输入数据(Reverse) …… 305
7.4.2 偷窥(Peeky) …… 308
7.5 seq2seq的应用 …… 313
7.5.1 聊天机器人 …… 314
7.5.2 算法学习 …… 315
7.5.3 自动图像描述 …… 316
7.6 小结 …… 318

第8章 Attention …… 321
8.1 Attention的结构 …… 321
8.1.1 seq2seq存在的问题 …… 322
8.1.2 编码器的改进 …… 323
8.1.3 解码器的改进① …… 325

8.1.4 解码器的改进② ······ 333
8.1.5 解码器的改进③ ······ 339
8.2 带Attention的seq2seq的实现 ······ 344
8.2.1 编码器的实现 ······ 344
8.2.2 解码器的实现 ······ 345
8.2.3 seq2seq的实现 ······ 347
8.3 Attention的评价 ······ 347
8.3.1 日期格式转换问题 ······ 348
8.3.2 带Attention的seq2seq的学习 ······ 349
8.3.3 Attention的可视化 ······ 353
8.4 关于Attention的其他话题 ······ 356
8.4.1 双向RNN ······ 356
8.4.2 Attention层的使用方法 ······ 358
8.4.3 seq2seq的深层化和skip connection ······ 360
8.5 Attention的应用 ······ 363
8.5.1 GNMT ······ 363
8.5.2 Transformer ······ 365
8.5.3 NTM ······ 369
8.6 小结 ······ 373

附录A sigmoid函数和tanh函数的导数 ······ 375
A.1 sigmoid函数 ······ 375
A.2 tanh函数 ······ 378
A.3 小结 ······ 380
附录B 运行WordNet ······ 381
B.1 NLTK的安装 ······ 381
B.2 使用WordNet获得同义词 ······ 382
B.3 WordNet和单词网络 ······ 384
B.4 基于WordNet的语义相似度 ······ 385
附录C GRU ······ 387
C.1 GRU的接口 ······ 387
C.2 GRU的计算图 ······ 388

后记 ······ 391
参考文献 ······ 395

译者序

翻译完《深度学习入门：基于Python的理论与实现》后不久，作者斋藤康毅又高产地出版了续作，专门讲解深度学习在自然语言处理中如何应用。翻到本书的前言，理查德·费曼所说的“凡我不能创造的，我就不能理解。”（“What I cannot create, I do not understand.”）这句话跃然纸上，而这句话在我为前作所写的译者序中也有提及。看来我和作者都比较认同这样一个观点：如果想真正弄清楚一件事情，就需要躬行实践。这或许也是一种缘分，所以我决定继续完成续作的翻译工作。

本书延续了前作的理念，但关注的应用领域不同：前作的内容以卷积神经网络和图像识别为主，而本书则侧重于循环神经网络和自然语言处理。本书详细介绍了单词向量、LSTM、seq2seq和Attention等自然语言处理中重要的深度学习技术。

当然，自然语言处理是一个综合性的研究领域，涉及语法、语义和语境等概念，有许多研究分支。除了深度学习这一研究范式之外，还有基于语言学、基于规则、基于机器学习的研究范式。本书涉及的单词含义、语言模型、文本生成只是其研究范围的一小部分。读者如果想更加全面地了解自然语言处理，还需要阅读更多相关资料。

前作出版后，很多读者在书评网站或者图灵社区反馈翻译得不错。不过，在翻译前作时，因为过于追求“信”，有时在语句的连贯性上稍有欠缺，个

别地方读起来甚至有些别扭。为此，本书的翻译在保证原文含义不变的情况下，更多地使用了符合中文表达习惯的表述方式，以求读者在阅读本书时，会有更佳的阅读体验。

本书的翻译由本人独立完成，特别感谢图灵的编辑对全书的审校。最后，由于译者水平有限，书中难免存在一些错误与疏漏。欢迎各位读者批评指正，将发现的问题通过图灵社区反馈给我们，以便我们在本书重印时进行改正。

陆宇杰

2020 年 2 月于上海

前言

凡我不能创造的，我就不能理解。

——理查德·费曼

深度学习正在深刻地改变这个世界。没有深度学习，智能手机的语音识别、Web 的实时翻译、汇率的预测都无从谈起。得益于深度学习，新药的研发、患者的诊断、汽车的自动驾驶都在渐渐成为现实。除此以外，几乎所有的高新技术背后都有深度学习的身影。今后，世界将因深度学习而前进得更远。

本书是《深度学习入门：基于 Python 的理论与实现》的续作，我们将在前作的基础上讨论深度学习的相关技术。特别是，本书将专注于自然语言处理和时序数据处理，使用深度学习挑战各种各样的任务。另外，本书继承了前作“从零开始创建”的理念，让读者充分体验深度学习相关的高新技术。

本书的理念

笔者认为，要深入理解深度学习（或者某个高新技术），“从零开始创建”的经验非常重要。从零开始创建，是指从自己可以理解的地方出发，在尽量不使用外部现成品的情况下，实现目标技术。本书的目标就是通过这样的过程来切实掌握深度学习（而不是仅停留在表面）。

说到底，要深入理解某项技术，至少应掌握创建它所需的知识和技能。因为本书要从零开始创建深度学习，所以我们会编写很多程序，进行很多实验。这个过程相当费时，有时还很费脑。不过，在这些费时的工作（甚至这样的工作本身）中包含许多对深入理解技术非常重要的精髓。如此获得的知识，对于使用既有库、阅读最前沿的论文、开发原创系统都非常有帮助。此外，最重要的是，一步一步地理解深度学习的结构和原理这件事本身就很有趣。

进入自然语言处理的世界

本书的主题是基于深度学习的自然语言处理。简言之，自然语言处理是让计算机理解我们日常所说的语言的技术。让计算机理解我们所说的语言是一件非常难的事情，同时也非常重要。实际上，自然语言处理技术已经极大地改变了我们的生活。Web 检索、机器翻译、语音助理，这些对世界产生了重大影响的技术在底层都用到了自然语言处理技术。

如上所述，我们的生活已经离不开自然语言处理技术。在这个领域中，深度学习也占有非常重要的地位。实际上，深度学习极大地改善了传统自然语言处理的性能。比如，谷歌的机器翻译性能就基于深度学习获得了显著提升。

本书将围绕自然语言处理和时序数据处理，来介绍深度学习的重要技巧，具体包括 word2vec、RNN、LSTM、GRU、seq2seq 和 Attention 等。本书将尽可能地用简洁的语言来解释这些技术，并通过实际创建它们来帮助读者加深理解。另外，通过实验，我们将实际感受到它们的潜力。

本书从深度学习的视角探索自然语言处理。全书一共 8 章，建议读者像读连载故事一样，从头开始顺序阅读。在遇到问题时，我们会先想办法解决问题，然后再改进解决办法。按照这种流程，我们以深度学习为武器，解决关于自然语言处理的各种问题。通过这次探险，希望读者能深入理解深度学习中的重要技巧，并体会到它们的有趣之处。

本书面向的读者

本书是《深度学习入门：基于 Python 的理论与实现》的续作，因此假定读者已经学习了前作的内容。但是，作为回顾，本书第 1 章会复习一下神经网络。因此，即便没有读过前作，只要具有神经网络和 Python 相关的知识，就也可以阅读本书。

为了让读者深入理解深度学习，本书将继承前作的理念，以“创建”“运行”为中心展开话题。不使用自己不理解的东西，只使用自己理解的东西，我们将坚定这样的立场，去探索深度学习和自然语言处理的世界。

为了明确本书的读者对象，这里将本书的内容和特征列举如下。

- 不依赖外部库，从零开始实现深度学习的程序
- 作为《深度学习入门：基于 Python 的理论与实现》的续作，围绕自然语言处理和时序数据处理中用到的深度学习技术进行讲解
- 提供可以运行的 Python 源代码，让读者能够方便地进行实验
- 尽可能地用简洁的语言和清晰的图示进行说明
- 虽然也会使用数学式，但更注重基于源代码进行解释
- 重视原理，比如“为什么这个方法更好?”“为什么这样有效?”“为什么这样有问题?”等

另外，这里将从本书中可以学到的技术列举如下。

- 基于 Python 的文本处理
- 深度学习之前的“单词”表示方法
- 用于获取单词向量的 word2vec（CBOW 模型和 skip-gram 模型）
- 加快大规模数据的训练速度的 Negative Sampling
- 处理时序数据的 RNN、LSTM 和 GRU
- 处理时序数据的误差反向传播法（Backpropagation Through Time）

- 进行文本生成的神经网络
- 将一个时序数据转化为另一个时序数据的 seq2seq
- 关注重要信息的 Attention

本书将以通俗易懂的方式详细解释这些技术，以便读者能在实现层面掌握它们。在讲解这些技术时，本书不会单单列举事实，而是会像故事连载一样展开叙述。

本书不面向的读者

明确本书不适合什么样的读者也很重要，为此，这里将本书不会涉及的内容列举如下。

- 不介绍深度学习相关的最新研究进展
- 不讨论 Caffe、TensorFlow 和 Chainer 等深度学习框架的使用方法
- 不提供深度学习理论层面的详细解释
- 不涉及图像识别、语音识别和强化学习等主题（本书主要关注自然语言处理）

如上所述，本书不涉及最新研究和理论细节。但是，读完本书之后，读者应该有能力去研究那些最新的论文或者自然语言处理相关的最前沿技术。

运行环境

本书提供了 Python 3 的源代码，读者可以自己动手实际运行这些源代码。通过边读代码边思考，并尝试自己想到的新思路，可以帮助自己巩固所学知识。本书中用到的源代码可以从以下网址下载：

https://www.ituring.com.cn/book/2678[①]

本书的目标是从零开始实现深度学习。因此，我们的方针是尽量不使用外部库，但是 NumPy 和 Matplotlib 这两个库例外。借助这两个库，我们可以高效地实现深度学习。

NumPy 是用于数值计算的库。该库提供了许多用于处理高级数学算法和数组（矩阵）的便捷方法。在本书的深度学习实现中，我们将使用这些便捷方法进行高效的实现。

Matplotlib 是用于绘图的库。使用 Matplotlib，可以将实验结果可视化，以直观地确认深度学习的学习过程。本书将使用这些库，来实现深度学习的算法。

另外，本书中的大部分源代码可以在普通计算机上运行，而且不会花费太多时间。但是，也是有一部分代码（特别是大型神经网络的学习）需要花费大量时间。为了加快这部分耗时代码的处理速度，本书还提供了能在 GPU 上运行的代码（机制）。这是通过一个名为 CuPy 的库实现的（CuPy 会在第 1 章介绍）。如果你有一台装有 NVIDIA GPU 的机器，通过安装 CuPy，可以在 GPU 上高速处理本书的部分代码。



本书使用如下编程语言和库。

- Python 3
- NumPy
- Matplotlib
- CuPy（可选）

① 请至本页面右侧的“随书下载”处下载本书源代码。另外，与本书内容相关的网址，均可在该页面下方的“相关文章”(https://www.ituring.com.cn/article/510370)处查询。——编者注

再次出发

技术已经进步到了可以轻松进行复制的时代。当下是一个可以轻松复制照片、视频、源代码和库的便捷世界。但是，无论技术多么发达，生活多么便利，经验都是无法轻松复制的。自己动手的经验、花时间思考的经验，都是无法复制的。而那些永恒的价值，正存在于这些无法复制的事物中。

前言到此结束，让我们再次踏上学习深度学习的旅途吧！

表述规则

本书在表述上采用如下规则。

粗体字（Bold）

用来表示新引入的术语、强调的要点以及关键短语。

等宽字（Constant Width）

用来表示下面这些信息：程序代码、命令、序列、组成元素、语句选项、分支、变量、属性、键值、函数、类型、类、命名空间、方法、模块、属性、参数、值、对象、事件、事件处理器、XML 标签、HTML 标签、宏、文件的内容、来自命令行的输出等。若在其他地方引用了以上这些内容（如变量、函数、关键字等），也会使用该格式标记。

等宽粗体字（**Constant Width Bold**）

用来表示用户输入的命令或文本信息。在强调代码的作用时也会使用该格式标记。

等宽斜体字（*Constant Width Italic*）

用来表示必须根据用户环境替换的字符串。



表示源代码的文件位置。



用来表示提示、启示以及某些值得深入研究的内容的补充信息。



表示程序库中存在的bug或经常会发生的问题等警告信息，引起读者对该处内容的注意。

读者意见与咨询

虽然笔者已经尽最大努力对本书的内容进行了检查与确认，但是仍不免在某些地方出现错误或者容易引起误解的表达及排版问题等。如果读者遇到这些问题，请及时告知，我们在本书重印时会将其改正。与此同时，也欢迎读者为本书将来的修订提出建议。本书编辑部的联系方式如下。

株式会社 O'Reilly Japan
电子邮件 japan@oreilly.co.jp

本书的主页地址如下。

https://www.ituring.com.cn/book/2678
https://www.oreilly.co.jp/books/9784873118369（日语）
https://github.com/oreilly-japan/deep-learning-from-scratch-2

关于 O'Reilly 的其他信息，可以访问下面的 O'Reilly 主页查看。

http://www.oreilly.com/（英语）

http://www.oreilly.co.jp/（日语）

电子书

扫描如下二维码，即可购买本书电子版。



第1章 神经网络的复习

用一种以上的方法认识一个事物，才能真正理解它。

——马文·明斯基（计算机科学家、认知科学家）

本书是《深度学习入门：基于 Python 的理论与实现》的续作，将进一步深入探索深度学习的可能性。本书和前作一样，不使用既有的库和框架，重视“从零开始创建”，通过亲自动手，来探寻深度学习相关技术的乐趣和深度。

本章我们将复习一下神经网络。也就是说，本章相当于前作的摘要。此外，本书更加重视效率，对前作中的部分代码规范进行了修改（比如，方法名和参数的命名方法等）。关于这一点，我们也会在本章进行确认。

1.1 数学和 Python 的复习

我们先来复习一下数学。具体来说，就是以神经网络的计算所需的向量、矩阵等为主题展开讨论。为了顺利地切入神经网络的实现，这里将一并展示相应的 Python 代码，特别是基于 NumPy 的代码。

1.1.1 向量和矩阵

在神经网络中，向量和矩阵（或者张量）随处可见。本节将对这些术语进行简单的整理，为读者阅读本书做准备。

我们从向量开始。向量是同时拥有大小和方向的量。向量可以表示为排成一排的数字集合，在 Python 实现中可以处理为一维数组。与此相对，矩阵是排成二维形状（长方阵）的数字集合。向量和矩阵的例子如图 1-1 所示。

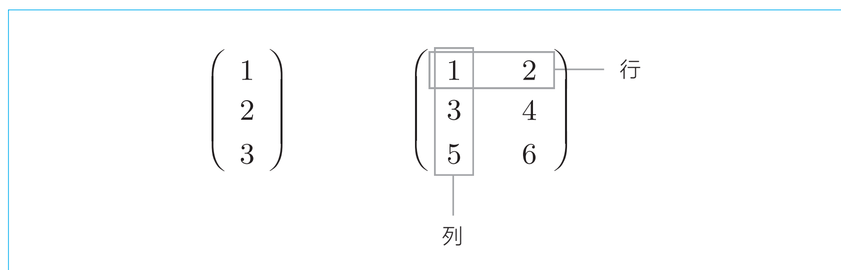


图 1-1　向量和矩阵的例子

如图 1-1 所示，向量和矩阵可以分别用一维数组和二维数组表示。另外，在矩阵中，将水平方向上的排列称为行（row），将垂直方向上的排列称为列（column）。因此，图 1-1 中的矩阵可以称为"3 行 2 列的矩阵"，记为"3×2 的矩阵"。



将向量和矩阵扩展到 N 维的数据集合，就是张量。

向量是一个简单的概念，请注意有两种方式表示向量。如图 1-2 所示，一种是在垂直方向上排列（列向量）的方法，另一种是在水平方向上排列（行向量）的方法。

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \qquad \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$$

列向量　　　　行向量

图 1-2　向量的表示方法

在数学和深度学习等许多领域，向量一般作为列向量处理。不过，考虑到实现层面的一致性，本书将向量作为行向量处理（每次都会注明是行向量）。此外，在数学式中写向量或矩阵时，会用 $\boldsymbol{x}$ 或 $\boldsymbol{W}$ 等粗体表示，以将它们与单个元素（标量）区分开。在源代码中，会用 `x` 或 `W` 这样的字体表示。



在 Python 的实现中，在将向量作为行向量处理的情况下，会将向量明确变形为水平方向上的矩阵。比如，当向量的元素个数是 N 时，将其处理为形状为 $1 \times N$ 的矩阵。我们后面会看一个具体的例子。

下面，我们使用 Python 的对话模式来生成向量和矩阵。当然，这里将使用处理矩阵的标准库 NumPy。

```
>>> import numpy as np

>>> x = np.array([1, 2, 3])
>>> x.__class__ # 输出类名
<class 'numpy.ndarray'>
>>> x.shape
(3,)
>>> x.ndim
1

>>> W = np.array([[1, 2, 3], [4, 5, 6]])
>>> W.shape
(2, 3)
>>> W.ndim
2
```

如上所示，可以使用 np.array() 方法生成向量或矩阵。该方法会生成 NumPy 的多维数组类 np.ndarray。np.ndarray 类有许多便捷的方法和实例变量。上面的例子中使用了实例变量 shape 和 ndim。shape 表示多维数组的形状，ndim 表示维数。从上面的结果可知，x 是一维数组，是一个元素个数为 3 的向量；而 W 是一个二维数组，是一个 2×3（2 行 3 列）的矩阵。

1.1.2 矩阵的对应元素的运算

前面我们把数字的集合组织为了向量或矩阵，现在利用它们进行一些简单的运算。首先，我们看一下“对应元素的运算”。顺便说一下，“对应元素的”的英文是“element-wise”。

```
>>> W = np.array([[1, 2, 3], [4, 5, 6]])
>>> X = np.array([[0, 1, 2], [3, 4, 5]])
>>> W + X
array([[ 1,  3,  5],
       [ 7,  9,  11]])
>>> W * X
array([[ 0,  2,  6],
       [12,  20, 30]])
```

这里对 NumPy 多维数组执行了 +、* 等运算。此时，运算是对应多维数组中的元素（独立）进行的，这就是 NumPy 数组中的对应元素的运算。

1.1.3 广播

在 NumPy 多维数组中，形状不同的数组之间也可以进行运算，比如下面这个计算。

```
>>> A = np.array([[1, 2], [3, 4]])
>>> A * 10
array([[10, 20],
       [30, 40]])
```

这个计算是一个 2 × 2 的矩阵 A 乘以标量 10。此时，如图 1-3 所示，标量 10 先被扩展为 2 × 2 的矩阵，之后进行对应元素的运算。这个灵巧的功能称为广播（broadcast）。

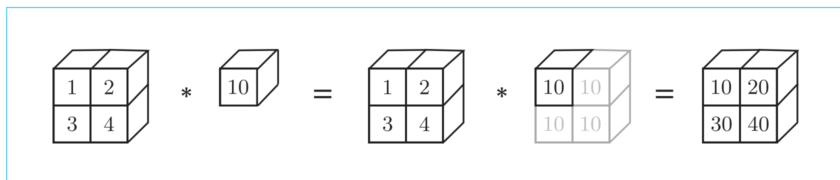


图 1-3　广播的例子 1：标量 10 被处理为 2 × 2 的矩阵

下面再看另一个广播的例子。

```
>>> A = np.array([[1, 2], [3, 4]])
>>> b = np.array([10, 20])
>>> A * b
array([[10, 40],
       [30, 80]])
```

在这个计算中，如图 1-4 所示，一维数组 b 被“灵巧地”扩展成了与二维数组 A 相同的形状。

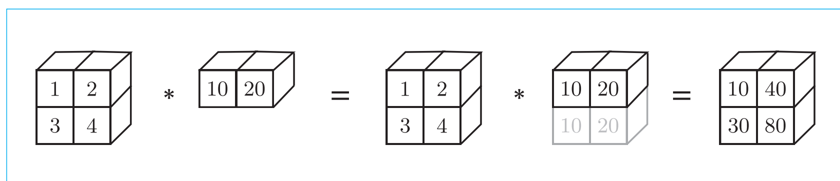


图 1-4　广播的例子 2

像这样，因为 NumPy 有广播功能，所以可以智能地执行不同形状的数组之间的运算。



为了使 NumPy 的广播功能生效，多维数组的形状需要满足几个规则。关于广播的详细规则，请参考文献 [1]。

1.1.4　向量内积和矩阵乘积

接着，我们来看一下向量内积和矩阵乘积的相关内容。首先，向量内积可以表示为

$$\boldsymbol{x}\cdot\boldsymbol{y}=x_1y_1+x_2y_2+\cdots+x_ny_n \tag{1.1}$$

这里假设有 $\boldsymbol{x}=(x_1,\cdots,x_n)$ 和 $\boldsymbol{y}=(y_1,\cdots,y_n)$ 两个向量。此时，如式 (1.1) 所示，向量内积是两个向量对应元素的乘积之和。



向量内积直观地表示了“两个向量在多大程度上指向同一方向”。如果限定向量的大小为 1，当两个向量完全指向同一方向时，它们的向量内积是 1。反之，如果两个向量方向相反，则内积为 −1。

下面再来看一下矩阵乘积。矩阵乘积可以按照图 1-5 所示的步骤计算。

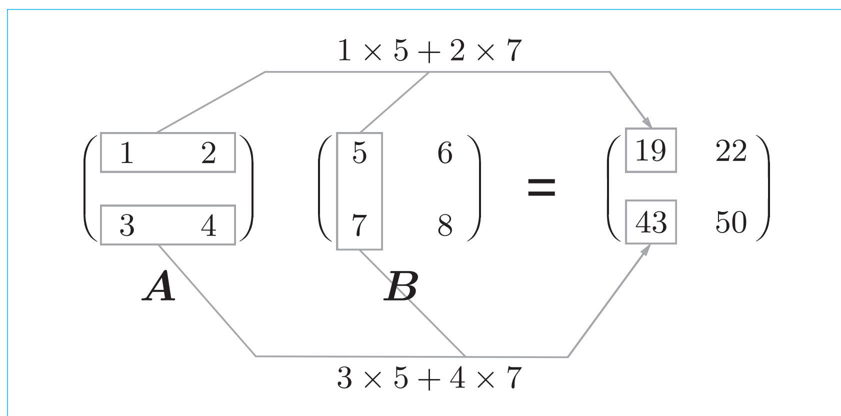


图 1-5　矩阵乘积的计算方法

如图 1-5 所示，矩阵乘积通过“左侧矩阵的行向量（水平方向）”和“右侧矩阵的列向量（垂直方向）”的内积（对应元素的乘积之和）计算得出。此时，计算结果保存在新矩阵的对应元素中。比如，$\boldsymbol{A}$ 的第 1 行和 $\boldsymbol{B}$ 的第 1 列的结果保存在第 1 行第 1 列的元素中，$\boldsymbol{A}$ 的第 2 行和 $\boldsymbol{B}$ 的第 1 列的结

果保存在第 2 行第 1 列的元素中。

现在，我们用 Python 实现一下向量内积和矩阵乘积。为此，可以利用 np.dot()。

```
# 向量内积
>>> a = np.array([1, 2, 3])
>>> b = np.array([4, 5, 6])
>>> np.dot(a, b)
32

# 矩阵乘积
>>> A = np.array([[1, 2], [3, 4]])
>>> B = np.array([[5, 6], [7, 8]])
>>> np.dot(A, B)
array([[19, 22],
       [43, 50]])
```

如上所示，向量内积和矩阵乘积中都使用了 np.dot()。当 np.dot(x, y) 的参数都是一维数组时，计算向量内积。当参数都是二维数组时，计算矩阵乘积。

除了这里看到的 np.dot() 方法外，NumPy 还有很多其他的进行矩阵计算的便捷方法。如果能熟练掌握这些方法，神经网络的实现就会更顺利。



熟能生巧

要掌握NumPy，实际动手练习是最有效的。比如，“100 numpy exercises”[2] 中准备了100道NumPy的练习题。如果你想积累NumPy经验，请挑战一下。

1.1.5　矩阵的形状检查

在使用矩阵和向量的计算中，很重要的一点就是要注意它们的形状。这里，我们留意着矩阵的形状，再来看一下矩阵乘积。前面已经介绍过了矩阵乘积的计算步骤，所以这里我们将重点放在图 1-6 所示的“形状检查”上。

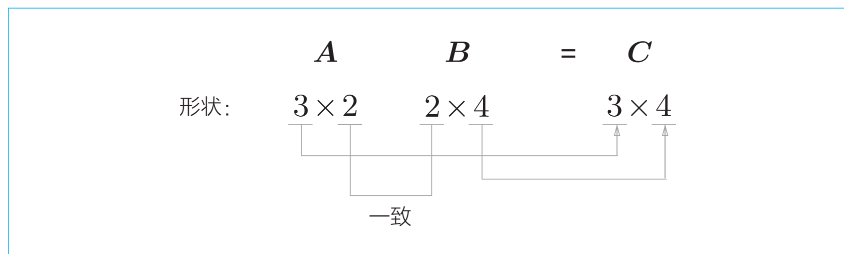


图 1-6 形状检查：在矩阵乘积中，要使对应维度的元素个数一致

图 1-6 展示了基于 3×2 的矩阵 $\boldsymbol{A}$ 和 2×4 的矩阵 $\boldsymbol{B}$ 生成 3×4 的矩阵 $\boldsymbol{C}$ 的示例。此时，如该图所示，需要对齐矩阵 $\boldsymbol{A}$ 和矩阵 $\boldsymbol{B}$ 的对应维度的元素个数。作为计算结果的矩阵 $\boldsymbol{C}$ 的形状由 $\boldsymbol{A}$ 的行数和 $\boldsymbol{B}$ 的列数组成。这就是矩阵的形状检查。



在矩阵乘积等计算中，注意矩阵的形状并观察其变化的形状检查非常重要。据此，神经网络的实现可以更顺利地进行。

1.2 神经网络的推理

现在我们开始复习神经网络。神经网络中进行的处理可以分为学习和推理两部分。本节将围绕神经网络的推理展开说明，而神经网络的学习会在下一节进行讨论。

1.2.1 神经网络的推理的全貌图

简单地说，神经网络就是一个函数。函数是将某些输入变换为某些输出的变换器，与此相同，神经网络也将输入变换为输出。

举个例子，我们来考虑输入二维数据、输出三维数据的函数。为了使用神经网络进行实现，需要在输入层准备 2 个神经元，在输出层准备 3 个神经元。然后，在隐藏层（中间层）放置若干神经元，这里我们放置 4 个神经元。

这样一来，我们的神经网络就可以画成图 1-7。

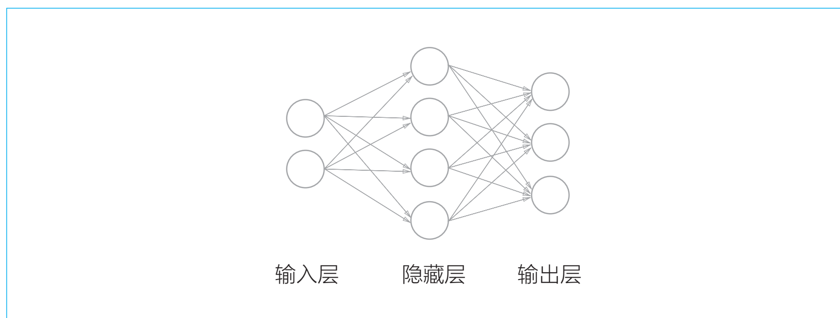


图 1-7 神经网络的例子

在图 1-7 中，用○表示神经元，用箭头表示它们的连接。此时，在箭头上有权重，这个权重和对应的神经元的值分别相乘，其和（严格地讲，是经过激活函数变换后的值）作为下一个神经元的输入。另外，此时还要加上一个不受前一层的神经元影响的常数，这个常数称为偏置。因为所有相邻的神经元之间都存在由箭头表示的连接，所以图 1-7 的神经网络称为全连接网络。



图 1-7 的网络一共包含 3 层，但有权重的层实际上是 2 层，本书中将这样的神经网络称为 2 层神经网络。因为图 1-7 的网络由 3 层组成，所以有的文献也称之为 3 层神经网络。

下面用数学式来表示图 1-7 的神经网络进行的计算。这里用 (x_1, x_2) 表示输入层的数据，用 w_{11} 和 w_{12} 表示权重，用 b_1 表示偏置。这样一来，图 1-7 中的隐藏层的第 1 个神经元就可以如下进行计算：

$$h_1 = x_1w_{11} + x_2w_{21} + b_1 \tag{1.2}$$

如式 (1.2) 所示，隐藏层的神经元是基于加权和计算出来的。之后，改变权重和偏置的值，根据神经元的个数，重复进行相应次数的式 (1.2) 的计算，这样就可以求出所有隐藏层神经元的值。

权重和偏置都有下标，这个下标的规则（为何将下标设为 11 或 12 等）并不是很重要，重要的是神经元是通过加权和计算的，并且可以通过矩阵乘积整体计算。实际上，基于全连接层的变换可以通过矩阵乘积如下进行整理：

$$(h_1, h_2, h_3, h_4) = (x_1, x_2)\begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{pmatrix} + (b_1, b_2, b_3, b_4) \tag{1.3}$$

这里，隐藏层的神经元被整理为 (h_1, h_2, h_3, h_4)，它可以看作 1×4 的矩阵（或者行向量）。另外，输入是 (x_1, x_2)，这是一个 1×2 的矩阵。再者，权重是 2×4 的矩阵，偏置是 1×4 的矩阵。这样一来，式 (1.3) 可以如下进行简化：

$$\boldsymbol{h} = \boldsymbol{x}\boldsymbol{W} + \boldsymbol{b} \tag{1.4}$$

这里，输入是 $\boldsymbol{x}$，隐藏层的神经元是 $\boldsymbol{h}$，权重是 $\boldsymbol{W}$，偏置是 $\boldsymbol{b}$，这些都是矩阵。此时，留意式 (1.4) 的矩阵形状，可知进行了如图 1-8 所示的变换。

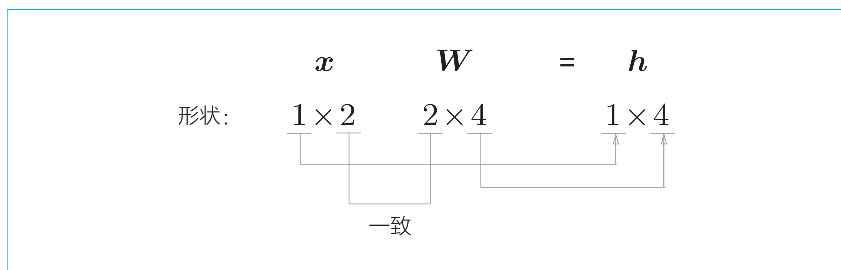


图 1-8 形状检查：确认对应维度的元素个数一致（省略偏置）

如图 1-8 所示，在矩阵乘积中，要使对应维度的元素个数一致。通过像这样观察矩阵的形状，可以确认变换是否正确。



在矩阵乘积的计算中，形状检查非常重要。据此，可以判断计算是否正确（至少可以判断计算是否成立）。

这样一来，我们就可以利用矩阵来整体计算全连接层的变换。不过，这里进行的变换只针对单笔样本数据（输入数据）。在神经网络领域，我们会同时对多笔样本数据（称为 mini-batch，小批量）进行推理和学习。因此，我们将单独的样本数据保存在矩阵 $\boldsymbol{x}$ 的各行中。假设要将 N 笔样本数据作为 mini-batch 整体处理，关注矩阵的形状，其变换如图 1-9 所示。

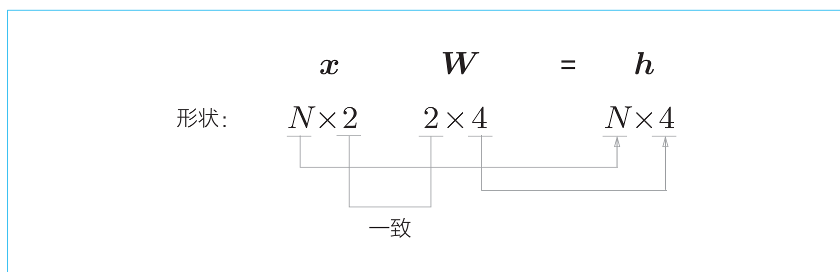


图 1-9　形状检查：mini-batch 版的矩阵乘积（省略偏置）

如图 1-9 所示，根据形状检查，可知各 mini-batch 被正确地进行了变换。此时，N 笔样本数据整体由全连接层进行变换，隐藏层的 N 个神经元被整体计算出来。现在，我们用 Python 写出 mini-batch 版的全连接层变换。

```
>>> import numpy as np
>>> W1 = np.random.randn(2, 4)  # 权重
>>> b1 = np.random.randn(4)     # 偏置
>>> x = np.random.randn(10, 2)  # 输入
>>> h = np.dot(x, W1) + b1
```

在这个例子中，10 笔样本数据分别由全连接层进行变换。此时，`x` 的第 1 个维度对应于各笔样本数据。比如，`x[0]` 是第 0 笔输入数据，`x[1]` 是第 1 笔输入数据……类似地，`h[0]` 是第 0 笔数据的隐藏层的神经元，`h[1]` 是第 1 笔数据的隐藏层的神经元，以此类推。



在上面的代码中，偏置 `b1` 的加法运算会触发广播功能。`b1` 的形状是 `(4,)`，它会被自动复制，变成 `(10, 4)` 的形状。

全连接层的变换是线性变换。激活函数赋予它“非线性”的效果。严格地讲，使用非线性的激活函数，可以增强神经网络的表现力。激活函数有很多种，这里我们使用式 (1.5) 的 sigmoid 函数（sigmoid function）：

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \tag{1.5}$$

如图 1-10 所示，sigmoid 函数呈 S 形曲线。

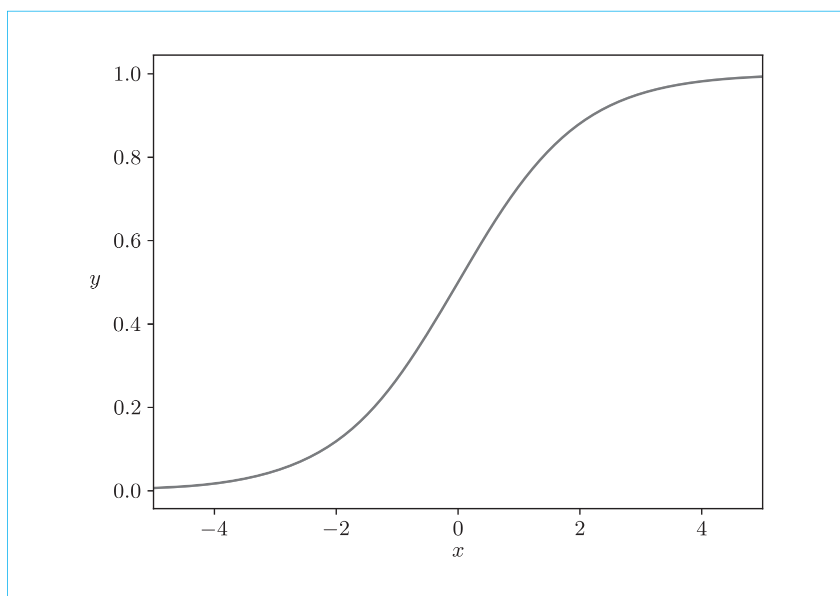


图1-10　sigmoid函数的图像

sigmoid 函数接收任意大小的实数，输出 0 ~ 1 的实数。现在我们用 Python 来实现这个 sigmoid 函数。

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

这是式 (1.5) 的直接实现，应该没有特别难的地方。现在，我们使用这个 sigmoid 函数，来变换刚才的隐藏层的神经元。

```
>>> a = sigmoid(h)
```

基于 sigmoid 函数，可以进行非线性变换。然后，再用另一个全连接层来变换这个激活函数的输出 a（也称为 activation）。这里，因为隐藏层有 4 个神经元，输出层有 3 个神经元，所以全连接层使用的权重矩阵的形状必须设置为 4×3，这样就可以获得输出层的神经元。以上就是神经网络的推理。现在我们用 Python 将这一段内容总结如下。

```
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

x = np.random.randn(10, 2)
W1 = np.random.randn(2, 4)
b1 = np.random.randn(4)
W2 = np.random.randn(4, 3)
b2 = np.random.randn(3)

h = np.dot(x, W1) + b1
a = sigmoid(h)
s = np.dot(a, W2) + b2
```

这里，x 的形状是 (10, 2)，表示 10 笔二维数据组织为了 1 个 mini-batch。最终输出的 s 的形状是 (10, 3)。同样，这意味着 10 笔数据一起被处理了，每笔数据都被变换为了三维数据。

上面的神经网络输出了三维数据。因此，使用各个维度的值，可以分为 3 个类别。在这种情况下，输出的三维向量的各个维度对应于各个类的“得分”（第 1 个神经元是第 1 个类别，第 2 个神经元是第 2 个类别……）。在实际进行分类时，寻找输出层神经元的最大值，将与该神经元对应的类别作为结果。



得分是计算概率之前的值。得分越高，这个神经元对应的类别的概率也越高。后面我们会看到，通过把得分输入 Softmax 函数，可以获得概率。

以上就是神经网络的推理部分的实现。接下来，我们使用Python的类，将这些处理实现为层。

1.2.2 层的类化及正向传播的实现

现在，我们将神经网络进行的处理实现为层。这里将全连接层的变换实现为 Affine 层，将 sigmoid 函数的变换实现为 Sigmoid 层。因为全连接层的变换相当于几何学领域的仿射变换，所以称为 Affine 层。另外，将各个层实现为 Python 的类，将主要的变换实现为类的 `forward()` 方法。



神经网络的推理所进行的处理相当于神经网络的正向传播。顾名思义，正向传播是从输入层到输出层的传播。此时，构成神经网络的各层从输入向输出方向按顺序传播处理结果。之后我们会进行神经网络的学习，那时会按与正向传播相反的顺序传播数据（梯度），所以称为反向传播。

神经网络中有各种各样的层，我们将其实现为 Python 的类。通过这种模块化，可以像搭建乐高积木一样构建网络。本书在实现这些层时，制定以下“代码规范”。

- 所有的层都有 `forward()` 方法和 `backward()` 方法
- 所有的层都有 `params` 和 `grads` 实例变量

简单说明一下这个代码规范。首先，`forward()` 方法和 `backward()` 方法分别对应正向传播和反向传播。其次，`params` 使用列表保存权重和偏置等参数（参数可能有多个，所以用列表保存）。`grads` 以与 `params` 中的参数对应的形式，使用列表保存各个参数的梯度（后述）。这就是本书的代码规范。



遵循上述代码规范，代码看上去会更清晰。我们后面会说明为什么要遵循这样的规范，以及它的有效性。

因为这里只考虑正向传播，所以我们仅关注代码规范中的以下两点：一是在层中实现 forward() 方法；二是将参数整理到实例变量 params 中。我们基于这样的代码规范来实现层，首先实现 Sigmoid 层，如下所示（✏ ch01/forward_net.py）。

```
import numpy as np

class Sigmoid:
    def __init__(self):
        self.params = []

    def forward(self, x):
        return 1 / (1 + np.exp(-x))
```

如上所示，sigmoid 函数被实现为一个类，主变换处理被实现为 forward(x) 方法。这里，因为 Sigmoid 层没有需要学习的参数，所以使用空列表来初始化实例变量 params。下面，我们接着来看一下全连接层 Affine 层的实现，如下所示（✏ ch01/forward_net.py）。

```
class Affine:
    def __init__(self, W, b):
        self.params = [W, b]

    def forward(self, x):
        W, b = self.params
        out = np.dot(x, W) + b
        return out
```

Affine 层在初始化时接收权重和偏置。此时，Affine 层的参数是权重和偏置（在神经网络的学习时，这两个参数随时被更新）。因此，我们使用列表将这两个参数保存在实例变量 params 中。然后，实现基于 forward(x) 的正向传播的处理。



根据本书的代码规范，所有的层都需要在实例变量 params 中保存要学习的参数。因此，可以很方便地将神经网络的全部参数整理在一起，参数的更新操作、在文件中保存参数的操作都会变得更容易。

现在，我们使用上面实现的层来实现神经网络的推理处理。这里实现如图 1-11 所示的层结构的神经网络。

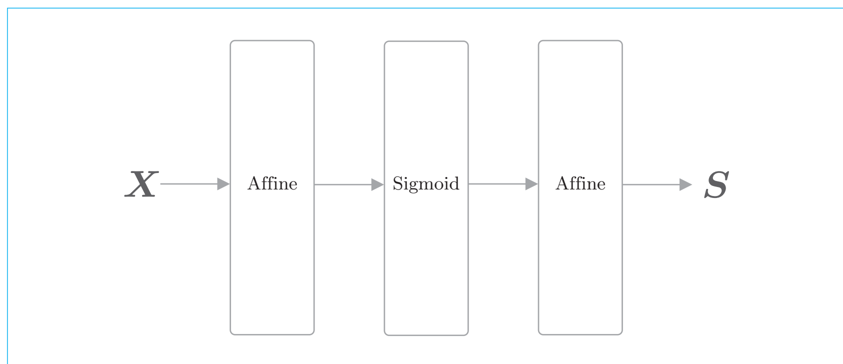


图 1-11 要实现的神经网络的层结构

如图 1-11 所示，输入 $\boldsymbol{X}$ 经由 Affine 层、Sigmoid 层和 Affine 层后输出得分 $\boldsymbol{S}$。我们将这个神经网络实现为名为 `TwoLayerNet` 的类，将主推理处理实现为 `predict(x)` 方法。



之前，我们在用图表示神经网络时，使用的是像图 1-7 那样的“神经元视角”的图。与此相对，图 1-11 是“层视角”的图。

`TwoLayerNet` 的代码如下所示（✏ `ch01/forward_net.py`）。

```
class TwoLayerNet:
    def __init__(self, input_size, hidden_size, output_size):
        I, H, O = input_size, hidden_size, output_size

        # 初始化权重和偏置
        W1 = np.random.randn(I, H)
        b1 = np.random.randn(H)
        W2 = np.random.randn(H, O)
        b2 = np.random.randn(O)
```

```
        # 生成层
        self.layers = [
            Affine(W1, b1),
            Sigmoid(),
            Affine(W2, b2)
        ]

        # 将所有的权重整理到列表中
        self.params = []
        for layer in self.layers:
            self.params += layer.params

    def predict(self, x):
        for layer in self.layers:
            x = layer.forward(x)
        return x
```

在这个类的初始化方法中，首先对权重进行初始化，生成 3 个层。然后，将要学习的权重参数一起保存在 params 列表中。这里，因为各个层的实例变量 params 中都保存了学习参数，所以只需要将它们拼接起来即可。这样一来，TwoLayerNet 的 params 变量中就保存了所有的学习参数。像这样，通过将参数整理到一个列表中，可以很轻松地进行参数的更新和保存。

此外，Python 中可以使用 + 运算符进行列表之间的拼接。下面是一个简单的例子。

```
>>> a = ['A', 'B']
>>> a += ['C', 'D']
>>> a
['A', 'B', 'C', 'D']
```

如上所示，通过列表之间的加法将列表拼接了起来。在上面的 TwoLayerNet 的实现中，通过将各个层的 params 列表加起来，从而将全部学习参数整理到了一个列表中。现在，我们使用 TwoLayerNet 类进行神经网络的推理。

```
x = np.random.randn(10, 2)
model = TwoLayerNet(2, 4, 3)
s = model.predict(x)
```

这样就可以求出输入数据 x 的得分 s 了。像这样，通过将层实现为类，可以轻松实现神经网络。另外，因为要学习的参数被汇总在 `model.params` 中，所以之后进行神经网络的学习会更加容易。

1.3　神经网络的学习

不进行神经网络的学习，就做不到“好的推理”。因此，常规的流程是，首先进行学习，然后再利用学习好的参数进行推理。所谓推理，就是对上一节介绍的多类别分类等问题给出回答的任务。而神经网络的学习的任务是寻找最优参数。本节我们就来研究神经网络的学习。

1.3.1　损失函数

在神经网络的学习中，为了知道学习进行得如何，需要一个指标。这个指标通常称为损失（loss）。损失指示学习阶段中某个时间点的神经网络的性能。基于监督数据（学习阶段获得的正确解数据）和神经网络的预测结果，将模型的恶劣程度作为标量（单一数值）计算出来，得到的就是损失。

计算神经网络的损失要使用损失函数（loss function）。进行多类别分类的神经网络通常使用交叉熵误差（cross entropy error）作为损失函数。此时，交叉熵误差由神经网络输出的各类别的概率和监督标签求得。

现在，我们来求一下之前一直在研究的那个神经网络的损失。这里，我们将 Softmax 层和 Cross Entropy Error 层新添加到网络中。用 Softmax 层求 Softmax 函数的值，用 Cross Entropy Error 层求交叉熵误差。如果基于“层视角”来绘制此时的网络结构，则如图 1-12 所示。

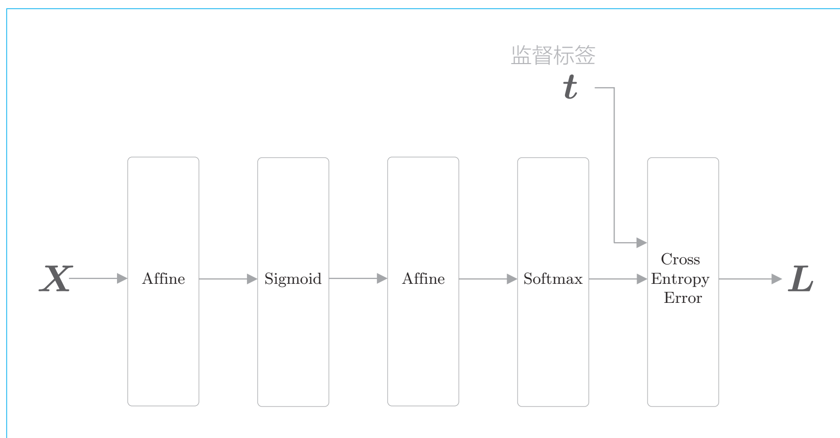


图 1-12 使用了损失函数的神经网络的层结构

在图 1-12 中，$\boldsymbol{X}$ 是输入数据，$\boldsymbol{t}$ 是监督标签，L 是损失。此时，Softmax 层的输出是概率，该概率和监督标签被输入 Cross Entropy Error 层。

下面，我们来介绍一下 Softmax 函数和交叉熵误差。首先，Softmax 函数可由下式表示：

$$y_k = \frac{\exp(s_k)}{\sum_{i=1}^{n} \exp(s_i)} \tag{1.6}$$

式 (1.6) 是当输出总共有 n 个时，计算第 k 个输出 y_k 时的算式。这个 y_k 是对应于第 k 个类别的 Softmax 函数的输出。如式 (1.6) 所示，Softmax 函数的分子是得分 s_k 的指数函数，分母是所有输入信号的指数函数的和。

Softmax 函数输出的各个元素是 0.0 ~ 1.0 的实数。另外，如果将这些元素全部加起来，则和为 1。因此，Softmax 的输出可以解释为概率。之后，这个概率会被输入交叉熵误差。此时，交叉熵误差可由下式表示：

$$L = -\sum_{k} t_k \log y_k \tag{1.7}$$

这里，t_k 是对应于第 k 个类别的监督标签。log 是以纳皮尔数 e 为底的对数（严格地说，应该记为 $\log_e$）。监督标签以 one-hot 向量的形式表示，比如 $\boldsymbol{t}=(0,0,1)$。



one-hot 向量是一个元素为 1，其他元素为 0 的向量。因为元素 1 对应正确解的类，所以式 (1.7) 实际上只是在计算正确解标签为 1 的元素所对应的输出的自然对数 (log)。

另外，在考虑了 mini-batch 处理的情况下，交叉熵误差可以由下式表示：

$$L=-\frac{1}{N}\sum_{n}\sum_{k}t_{nk}\log y_{nk} \tag{1.8}$$

这里假设数据有 N 笔，t_{nk} 表示第 n 笔数据的第 k 维元素的值，y_{nk} 表示神经网络的输出，t_{nk} 表示监督标签。

式 (1.8) 看上去有些复杂，其实只是将表示单笔数据的损失函数的式 (1.7) 扩展到了 N 笔数据的情况。用式 (1.8) 除以 N，可以求单笔数据的平均损失。通过这样的平均化，无论 mini-batch 的大小如何，都始终可以获得一致的指标。

本书将计算 Softmax 函数和交叉熵误差的层实现为 Softmax with Loss 层（通过整合这两个层，反向传播的计算会变简单）。因此，学习阶段的神经网络具有如图 1-13 所示的层结构。

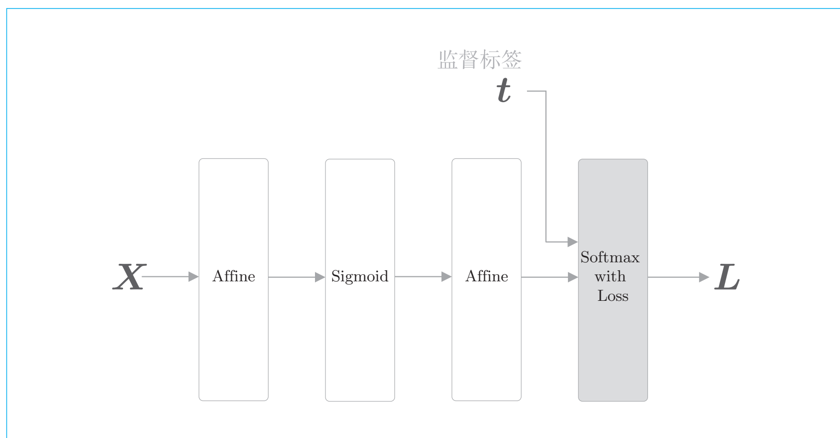


图1-13　使用Softmax with Loss层输出损失

如图 1-13 所示，本书使用 Softmax with Loss 层。这里我们省略对其实现的说明，代码在 `common/layers.py` 中，感兴趣的读者可以参考。此外，前作《深度学习入门：基于 Python 的理论与实现》的 4.2 节中也详细介绍了 Softmax with Loss 层。

1.3.2　导数和梯度

神经网络的学习的目标是找到损失尽可能小的参数。此时，导数和梯度非常重要。这里我们来简单说明一下导数和梯度。

现在，假设有一个函数 $y = f(x)$。此时，y 关于 x 的导数记为 $\frac{\mathrm{d}y}{\mathrm{d}x}$。这个 $\frac{\mathrm{d}y}{\mathrm{d}x}$ 的意思是变化程度，具体来说，就是 x 的微小（严格地讲，“微小”为无限小）变化会导致 y 发生多大程度的变化。

比如函数 $y = x^2$，其导数可以解析性地求出，即 $\frac{\mathrm{d}y}{\mathrm{d}x} = 2x$。这个导数结果表示 x 在各处的变化程度。实际上，如图 1-14 所示，它相当于函数的斜率。

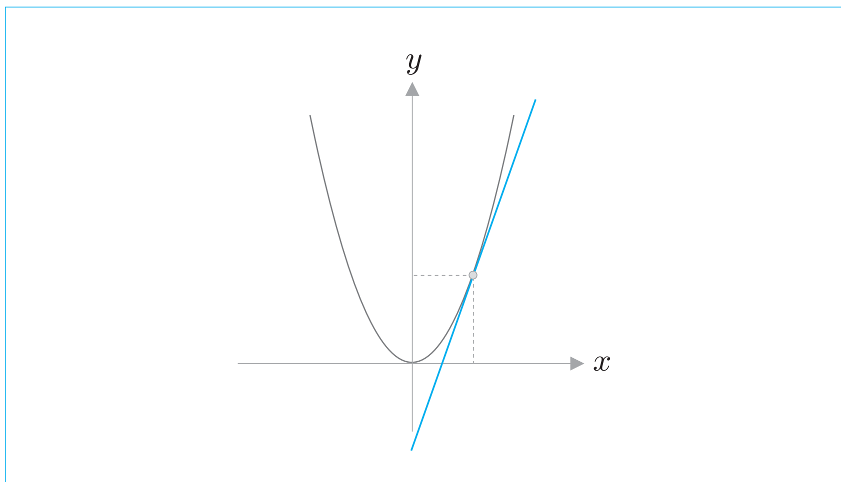


图1-14 $y = x^2$的导数表示x在各处的斜率

在图 1-14 中，我们求了关于 x 这一个变量的导数，其实同样可以求关于多个变量（多变量）的导数[①]。假设有函数 $L = f(\boldsymbol{x})$，其中 L 是标量，$\boldsymbol{x}$ 是向量。此时，L 关于 x_i（$\boldsymbol{x}$ 的第 i 个元素）的导数可以写成 $\frac{\partial L}{\partial x_i}$。另外，也可以求关于向量的其他元素的导数，我们将其整理如下：

$$\frac{\partial L}{\partial \boldsymbol{x}} = \left(\frac{\partial L}{\partial x_1}, \frac{\partial L}{\partial x_2}, \cdots, \frac{\partial L}{\partial x_n} \right) \tag{1.9}$$

像这样，将关于向量各个元素的导数罗列到一起，就得到了梯度（gradient）。

另外，矩阵也可以像向量一样求梯度。假设 $\boldsymbol{W}$ 是一个 $m \times n$ 的矩阵，则函数 $L = g(\boldsymbol{W})$ 的梯度如下所示：

$$\frac{\partial L}{\partial \boldsymbol{W}} = \begin{pmatrix} \frac{\partial L}{\partial W_{11}} & \cdots & \frac{\partial L}{\partial W_{1n}} \\ \vdots & \ddots & \\ \frac{\partial L}{\partial W_{m1}} & & \frac{\partial L}{\partial W_{mn}} \end{pmatrix} \tag{1.10}$$

① 从严格意义上讲，对多变量函数的某个变量求得的导数称为偏导数。本书考虑到易读性，在不影响理解的情况下，统一使用“导数”一词。——译者注

如式 (1.10) 所示，L 关于 $\boldsymbol{W}$ 的梯度可以写成矩阵（准确地说，矩阵的梯度的定义如上所示）。这里的重点是，$\boldsymbol{W}$ 和 $\frac{\partial L}{\partial \boldsymbol{W}}$ 具有相同的形状。利用“矩阵和其梯度具有相同形状”这一性质，可以轻松地进行参数的更新和链式法则（后述）的实现。



严格地说，本书使用的“梯度”一词与数学中的“梯度”是不同的。数学中的梯度仅限于关于向量的导数。而在深度学习领域，一般也会定义关于矩阵和张量的导数，称为“梯度”。

1.3.3　链式法则

学习阶段的神经网络在给定学习数据后会输出损失。这里我们想得到的是损失关于各个参数的梯度。只要得到了它们的梯度，就可以使用这些梯度进行参数更新。那么，神经网络的梯度怎么求呢？这就轮到误差反向传播法出场了。

理解误差反向传播法的关键是链式法则。链式法则是复合函数的求导法则，其中复合函数是由多个函数构成的函数。

现在，我们来学习链式法则。这里考虑 $y = f(x)$ 和 $z = g(y)$ 这两个函数。如 $z = g(f(x))$ 所示，最终的输出 z 由两个函数计算而来。此时，z 关于 x 的导数可以按下式求得：

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x} \tag{1.11}$$

如式 (1.11) 所示，z 关于 x 的导数由 $y = f(x)$ 的导数和 $z = g(y)$ 的导数之积求得，这就是链式法则。链式法则的重要之处在于，无论我们要处理的函数有多复杂（无论复合了多少个函数），都可以根据它们各自的导数来求复合函数的导数。也就是说，只要能够计算各个函数的局部的导数，就能基于它们的积计算最终的整体的导数。



可以认为神经网络是由多个函数复合而成的。误差反向传播法会充分利用链式法则来求关于多个函数（神经网络）的梯度。

1.3.4　计算图

下面，我们将研究误差反向传播法。不过在此之前，作为准备工作，我们先来介绍一下计算图的相关内容。计算图是计算过程的图形表示。图 1-15 所示为计算图的一个例子。

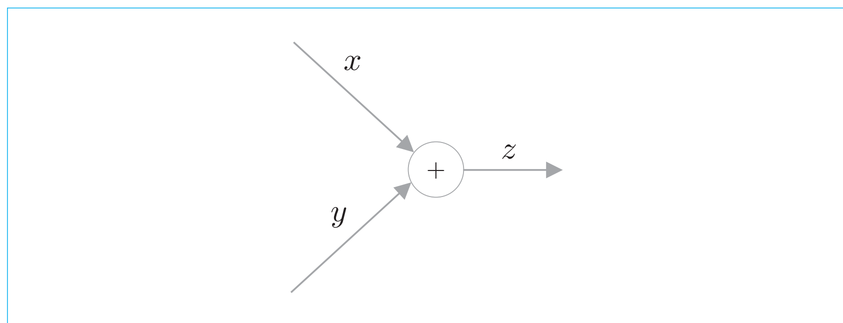


图 1-15　$z = x + y$ 的计算图

如图 1-15 所示，计算图通过节点和箭头来表示。这里，“+”表示加法，变量 x 和 y 写在各自的箭头上。像这样，在计算图中，用节点表示计算，处理结果有序（本例中是从左到右）流动。这就是计算图的正向传播。

使用计算图，可以直观地把握计算过程。另外，这样也可以直观地求梯度。这里重要的是，梯度沿与正向传播相反的方向传播，这个反方向的传播称为**反向传播**。

这里我想先说明一下反向传播的全貌。虽然我们处理的是 $z = x + y$ 这一计算，但是在该计算的前后，还存在其他的“某种计算”（图 1-16）。另外，假设最终输出的是标量 L（在神经网络的学习阶段，计算图的最终输出是损失，它是一个标量）。

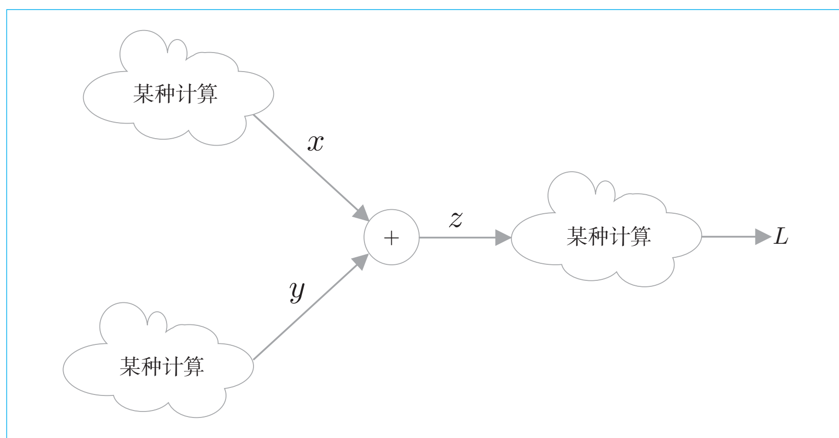


图 1-16　加法节点构成“复杂计算”的一部分

我们的目标是求 L 关于各个变量的导数（梯度）。这样一来，计算图的反向传播就可以绘制成图 1-17。

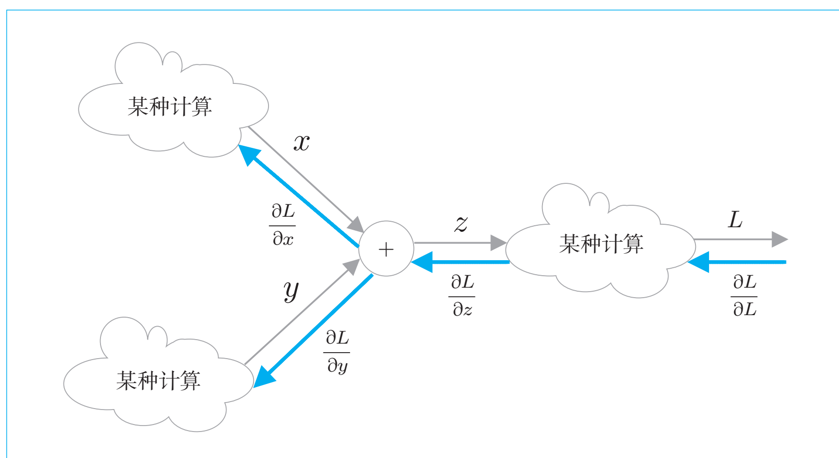


图 1-17　计算图的反向传播

如图 1-17 所示，反向传播用蓝色的粗箭头表示，在箭头的下方标注传播的值。此时，传播的值是指最终的输出 L 关于各个变量的导数。在这个例子中，关于 z 的导数是 $\frac{\partial L}{\partial z}$，关于 x 和 y 的导数分别是 $\frac{\partial L}{\partial x}$ 和 $\frac{\partial L}{\partial y}$。

接着，该链式法则出场了。根据刚才复习的链式法则，反向传播中流动的导数的值是根据从上游（输出侧）传来的导数和各个运算节点的局部导数之积求得的。因此，在上面的例子中，$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$，$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$。

这里，我们来处理 $z = x + y$ 这个基于加法节点的运算。此时，分别解析性地求得 $\frac{\partial z}{\partial x} = 1$，$\frac{\partial z}{\partial y} = 1$。因此，如图 1-18 所示，加法节点将上游传来的值乘以 1，再将该梯度向下游传播。也就是说，只是原样地将从上游传来的梯度传播出去。

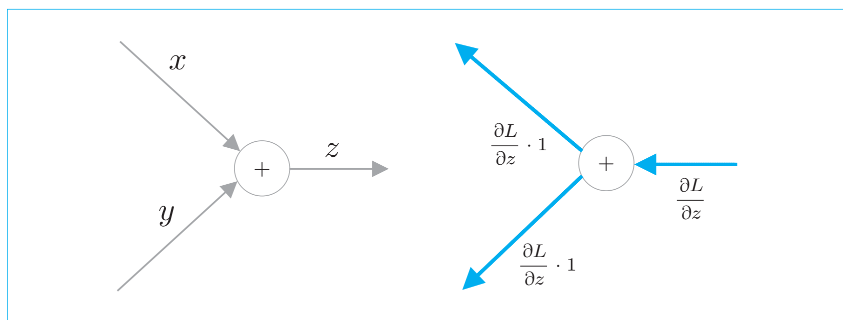


图 1-18　加法节点的正向传播（左图）和反向传播（右图）

像这样，计算图直观地表示了计算过程。另外，通过观察反向传播的梯度的流动，可以帮助我们理解反向传播的推导过程。

在构成计算图的运算节点中，除了这里见到的加法节点之外，还有很多其他的运算节点。下面，我们将介绍几个典型的运算节点。

1.3.4.1　乘法节点

乘法节点是 $z = x \times y$ 这样的计算。此时，导数可以分别求出，即 $\frac{\partial z}{\partial x} = y$ 和 $\frac{\partial z}{\partial y} = x$。因此，如图 1-19 所示，乘法节点的反向传播会将“上游传来的梯度”乘以“将正向传播时的输入替换后的值”。

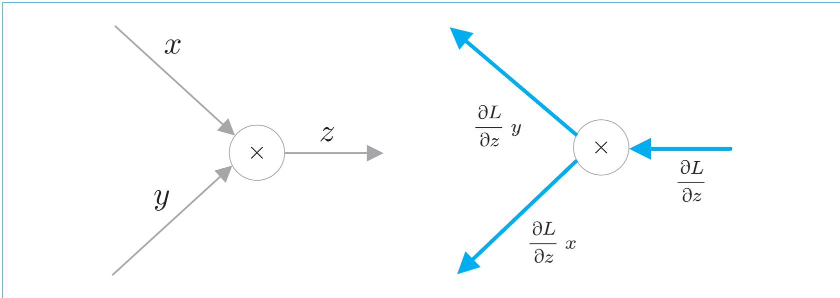


图 1-19 乘法节点的正向传播(左图)和反向传播(右图)

另外，在目前为止的加法节点和乘法节点的介绍中，流过节点的数据都是“单变量”。但是，不仅限于单变量，也可以是多变量（向量、矩阵或张量）。当张量流过加法节点（或者乘法节点）时，只需独立计算张量中的各个元素。也就是说，在这种情况下，张量的各个元素独立于其他元素进行对应元素的运算。

1.3.4.2 分支节点

如图 1-20 所示，分支节点是有分支的节点。

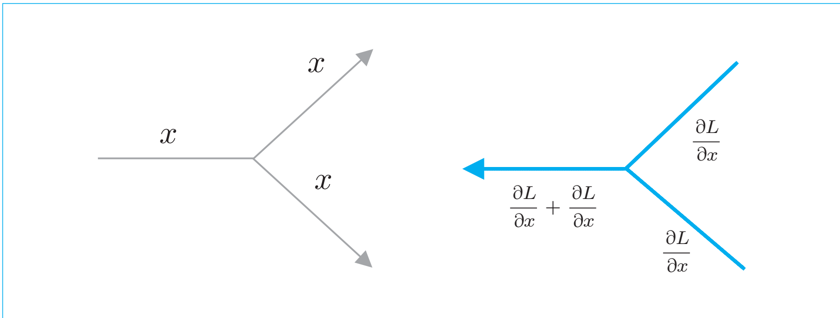


图 1-20 分支节点的正向传播(左图)和反向传播(右图)

严格来说，分支节点并没有节点，只有两根分开的线。此时，相同的值被复制并分叉。因此，分支节点也称为复制节点。如图 1-20 所示，它的反向传播是上游传来的梯度之和。

1.3.4.3 Repeat 节点

分支节点有两个分支，但也可以扩展为 N 个分支（副本），这里称为 Repeat 节点。现在，我们尝试用计算图绘制一个 Repeat 节点（图 1-21）。

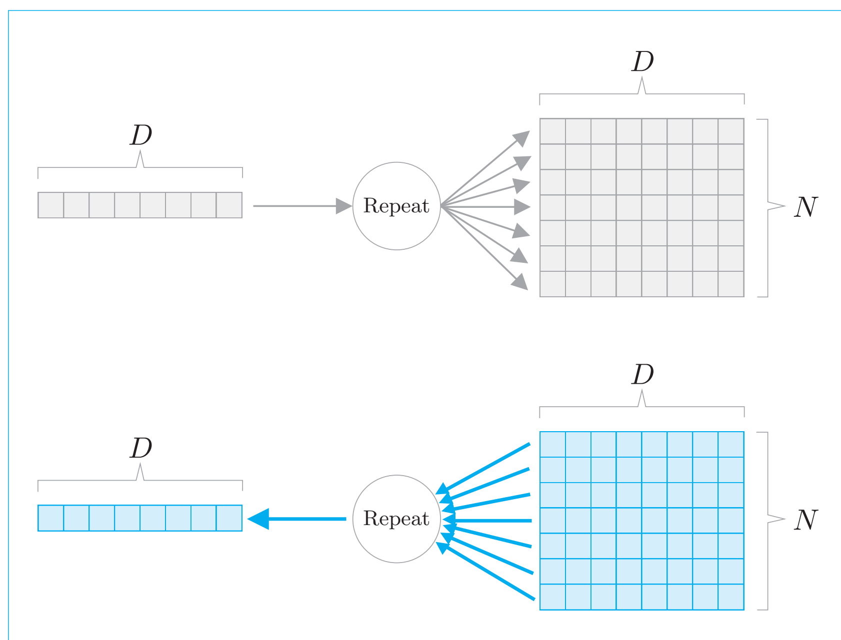


图 1-21 Repeat 节点的正向传播（上图）和反向传播（下图）

如图 1-21 所示，这个例子中将长度为 D 的数组复制了 N 份。因为这个 Repeat 节点可以视为 N 个分支节点，所以它的反向传播可以通过 N 个梯度的总和求出，如下所示。

```
>>> import numpy as np
>>> D, N = 8, 7
>>> x = np.random.randn(1, D)                # 输入
>>> y = np.repeat(x, N, axis=0)              # 正向传播

>>> dy = np.random.randn(N, D)               # 假设的梯度
>>> dx = np.sum(dy, axis=0, keepdims=True)   # 反向传播
```

这里通过 np.repeat() 方法进行元素的复制。上面的例子中将复制 N 次数组 x。通过指定 axis，可以指定沿哪个轴复制。因为反向传播时要计算总和，所以使用 NumPy 的 sum() 方法。此时，通过指定 axis 来指定对哪个轴求和。另外，通过指定 keepdims=True，可以维持二维数组的维数。在上面的例子中，当 keepdims=True 时，np.sum() 的结果的形状是 (1, D)；当 keepdims=False 时，形状是 (D,)。



NumPy 的广播会复制数组的元素。这可以通过 Repeat 节点来表示。

1.3.4.4 Sum 节点

Sum 节点是通用的加法节点。这里考虑对一个 $N \times D$ 的数组沿第 0 个轴求和。此时，Sum 节点的正向传播和反向传播如图 1-22 所示。

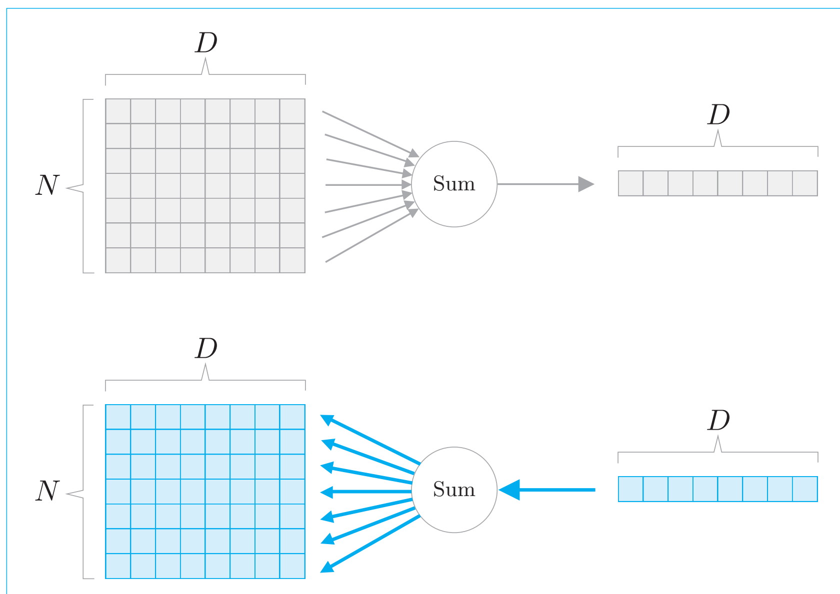


图 1-22　Sum 节点的正向传播（上图）和反向传播（下图）

如图 1-22 所示，Sum 节点的反向传播将上游传来的梯度分配到所有箭头上。这是加法节点的反向传播的自然扩展。下面，和 Repeat 节点一样，我们也来展示一下 Sum 节点的实现示例，如下所示。

```
>>> import numpy as np
>>> D, N = 8, 7
>>> x = np.random.randn(N, D)              # 输入
>>> y = np.sum(x, axis=0, keepdims=True)  # 正向传播

>>> dy = np.random.randn(1, D)             # 假设的梯度
>>> dx = np.repeat(dy, N, axis=0)          # 反向传播
```

如上所示，Sum 节点的正向传播通过 `np.sum()` 方法实现，反向传播通过 `np.repeat()` 方法实现。有趣的是，Sum 节点和 Repeat 节点存在逆向关系。所谓逆向关系，是指 Sum 节点的正向传播相当于 Repeat 节点的反向传播，Sum 节点的反向传播相当于 Repeat 节点的正向传播。

1.3.4.5 MatMul 节点

本书将矩阵乘积称为 MatMul 节点。MatMul 是 Matrix Multiply 的缩写。因为 MatMul 节点的反向传播稍微有些复杂，所以这里我们先进行一般性的介绍，再进行直观的解释。

为了解释 MatMul 节点，我们来考虑 $\boldsymbol{y} = \boldsymbol{x}\boldsymbol{W}$ 这个计算。这里，$\boldsymbol{x}$、$\boldsymbol{W}$、$\boldsymbol{y}$ 的形状分别是 $1 \times D$、$D \times H$、$1 \times H$（图 1-23）。

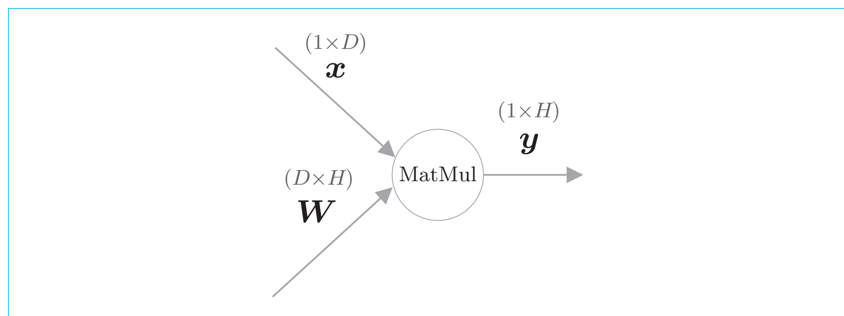


图 1-23 MatMul 节点的正向传播：矩阵的形状显示在各个变量的上方

此时，可以按如下方式求得关于 $\boldsymbol{x}$ 的第 i 个元素的导数 $\frac{\partial L}{\partial x_i}$。

$$\frac{\partial L}{\partial x_i} = \sum_j \frac{\partial L}{\partial y_j}\frac{\partial y_j}{\partial x_i} \tag{1.12}$$

式 (1.12) 的 $\frac{\partial L}{\partial x_i}$ 表示变化程度，即当 x_i 发生微小的变化时，L 会有多大程度的变化。如果此时改变 x_i，则向量 $\boldsymbol{y}$ 的所有元素都会发生变化。另外，因为 $\boldsymbol{y}$ 的各个元素会发生变化，所以最终 L 也会发生变化。因此，从 x_i 到 L 的链式法则的路径有多个，它们的和是 $\frac{\partial L}{\partial x_i}$。

式 (1.12) 仍可进一步简化。利用 $\frac{\partial y_j}{\partial x_i} = W_{ij}$，将其代入式 (1.12)：

$$\frac{\partial L}{\partial x_i} = \sum_j \frac{\partial L}{\partial y_j}\frac{\partial y_j}{\partial x_i} = \sum_j \frac{\partial L}{\partial y_j} W_{ij} \tag{1.13}$$

由式 (1.13) 可知，$\frac{\partial L}{\partial x_i}$ 由向量 $\frac{\partial L}{\partial \boldsymbol{y}}$ 和 $\boldsymbol{W}$ 的第 i 行向量的内积求得。从这个关系可以导出下式：

$$\frac{\partial L}{\partial \boldsymbol{x}} = \frac{\partial L}{\partial \boldsymbol{y}} \boldsymbol{W}^{\mathrm{T}} \tag{1.14}$$

如式 (1.14) 所示，$\frac{\partial L}{\partial \boldsymbol{x}}$ 可由矩阵乘积一次求得。这里，$\boldsymbol{W}^{\mathrm{T}}$ 的 T 表示转置矩阵。对式 (1.14) 进行形状检查，结果如图 1-24 所示。

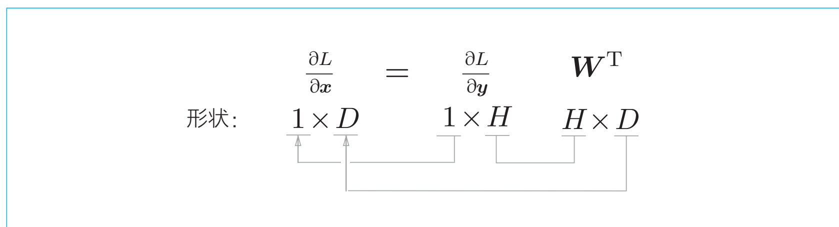


图 1-24　矩阵乘积的形状检查

如图 1-24 所示，矩阵形状的演变是正确的。由此，可以确认式 (1.14) 的计算是正确的。然后，我们可以反过来利用它（为了保持形状合规）来推

导出反向传播的数学式（及其实现）。为了说明这个方法，我们再次考虑矩阵乘积的计算 $\boldsymbol{y} = \boldsymbol{x}\boldsymbol{W}$。不过，这次考虑 mini-batch 处理，假设 $\boldsymbol{x}$ 中保存了 N 笔数据。此时，$\boldsymbol{x}$、$\boldsymbol{W}$、$\boldsymbol{y}$ 的形状分别是 $N \times D$、$D \times H$、$N \times H$，反向传播的计算图如图 1-25 所示。

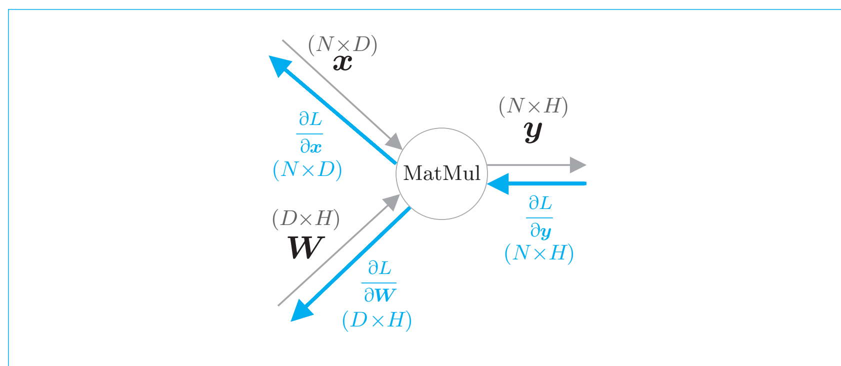


图 1-25　MatMul 节点的反向传播

那么，$\frac{\partial L}{\partial \boldsymbol{x}}$ 将如何计算呢？此时，和 $\frac{\partial L}{\partial \boldsymbol{x}}$ 相关的变量（矩阵）是上游传来的 $\frac{\partial L}{\partial \boldsymbol{y}}$ 和 $\boldsymbol{W}$。为什么说和 $\boldsymbol{W}$ 有关系呢？考虑到乘法的反向传播的话，就容易理解了。因为乘法的反向传播中使用了“将正向传播时的输入替换后的值”。同理，矩阵乘积的反向传播也使用“将正向传播时的输入替换后的矩阵”。之后，留意各个矩阵的形状求矩阵乘积，使它们的形状保持合规。如此，就可以导出矩阵乘积的反向传播，如图 1-26 所示。

如图 1-26 所示，通过确认矩阵的形状，可以推导矩阵乘积的反向传播的数学式。这样一来，我们就推导出了 MatMul 节点的反向传播。现在我们将 MatMul 节点实现为层，如下所示（✏ `common/layers.py`）。

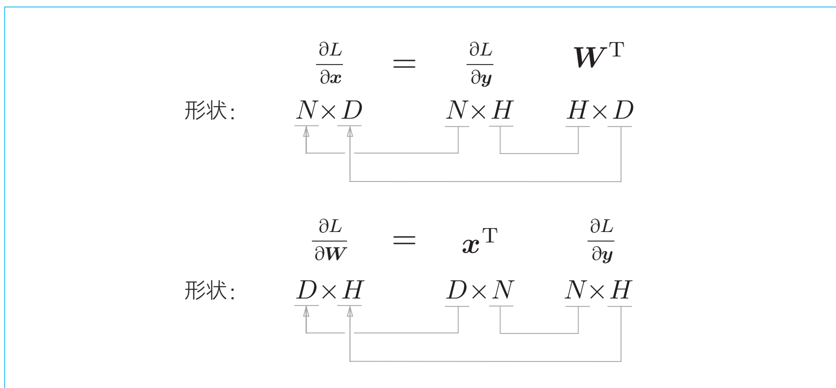


图 1-26　通过确认矩阵形状，推导反向传播的数学式

```
class MatMul:
    def __init__(self, W):
        self.params = [W]
        self.grads = [np.zeros_like(W)]
        self.x = None

    def forward(self, x):
        W, = self.params
        out = np.dot(x, W)
        self.x = x
        return out

    def backward(self, dout):
        W, = self.params
        dx = np.dot(dout, W.T)
        dW = np.dot(self.x.T, dout)
        self.grads[0][...] = dW
        return dx
```

MatMul 层在 params 中保存要学习的参数。另外，以与其对应的形式，将梯度保存在 grads 中。在反向传播时求 dx 和 dw，并在实例变量 grads 中设置权重的梯度。

另外，在设置梯度的值时，像 grads[0][...] = dW 这样，使用了省略号。

由此，可以固定 NumPy 数组的内存地址，覆盖 NumPy 数组的元素。



和省略号一样，这里也可以进行基于`grads[0] = dW`的赋值。不同的是，在使用省略号的情况下会覆盖掉NumPy数组。这是浅复制(shallow copy)和深复制(deep copy)的差异。`grads[0] = dW`的赋值相当于浅复制，`grads[0][...] = dW`的覆盖相当于深复制。

省略号的话题稍微有些复杂，我们举个例子来说明。假设有 a 和 b 两个 NumPy 数组。

```
>>> a = np.array([1, 2, 3])
>>> b = np.array([4, 5, 6])
```

这里，不管是 `a = b`，还是 `a[...] = b`，a 都被赋值 [4,5,6]。但是，此时 a 指向的内存地址不同。我们将内存（简化版）可视化，如图 1-27 所示。

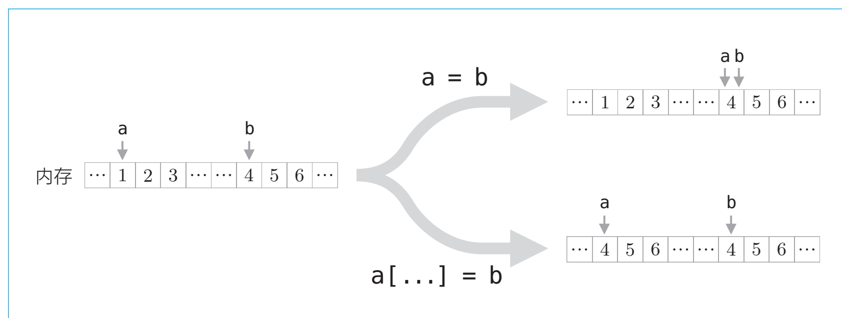


图 1-27　a=b 和 a[...]=b 的区别：使用省略号时数据被覆盖，变量指向的内存地址不变

如图 1-27 所示，在 `a = b` 的情况下，a 指向的内存地址和 b 一样。由于实际的数据（4,5,6）没有被复制，所以这可以说是浅复制。而在 `a[...] = b` 时，a 的内存地址保持不变，b 的元素被复制到 a 指向的内存上。这时，因为实际的数据被复制了，所以称为深复制。

由此可知，使用省略号可以固定变量的内存地址（在上面的例子中，a

的地址是固定的)。通过固定这个内存地址，实例变量 grads 的处理会变简单。



在 grads 列表中保存各个参数的梯度。此时，grads 列表中的各个元素是 NumPy 数组，仅在生成层时生成一次。然后，使用省略号，在不改变 NumPy 数组的内存地址的情况下覆盖数据。这样一来，将梯度汇总在一起的工作就只需要在开始时进行一次即可。

以上就是 MatMul 层的实现，代码在 common/layers.py 中。

1.3.5 梯度的推导和反向传播的实现

计算图的介绍结束了，下面我们来实现一些实用的层。这里，我们将实现 Sigmoid 层、全连接层 Affine 层和 Softmax with Loss 层。

1.3.5.1 Sigmoid 层

sigmoid 函数由 $y = \frac{1}{1+\exp(-x)}$ 表示，sigmoid 函数的导数由下式表示。

$$\frac{\partial y}{\partial x} = y(1-y) \tag{1.15}$$

根据式 (1.15)，Sigmoid 层的计算图可以绘制成图 1-28。这里，将输出侧的层传来的梯度（$\frac{\partial L}{\partial y}$）乘以 sigmoid 函数的导数（$\frac{\partial L}{\partial x}$），然后将这个值传递给输入侧的层。

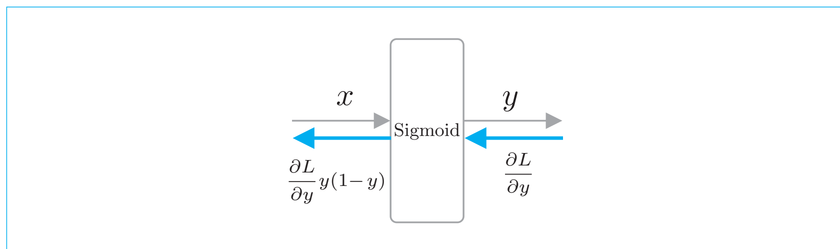


图 1-28 Sigmoid 层的计算图



这里，我们省略sigmoid函数的偏导数的推导过程。相关内容会在附录A中介绍，感兴趣的读者可以参考一下。

接下来，我们使用 Python 来实现 Sigmoid 层。参考图 1-28，可以像下面这样进行实现（✏ common/layers.py）。

```
class Sigmoid:
    def __init__(self):
        self.params, self.grads = [], []
        self.out = None

    def forward(self, x):
        out = 1 / (1 + np.exp(-x))
        self.out = out
        return out

    def backward(self, dout):
        dx = dout * (1.0 - self.out) * self.out
        return dx
```

这里将正向传播的输出保存在实例变量 out 中。然后，在反向传播中，使用这个 out 变量进行计算。

1.3.5.2　Affine 层

如前所示，我们通过 y = np.dot(x, W) + b 实现了 Affine 层的正向传播。此时，在偏置的加法中，使用了 NumPy 的广播功能。如果明示这一点，则 Affine 层的计算图如图 1-29 所示。

如图 1-29 所示，通过 MatMul 节点进行矩阵乘积的计算。偏置被 Repeat 节点复制，然后进行加法运算（可以认为 NumPy 的广播功能在内部进行了 Repeat 节点的计算）。下面是 Affine 层的实现（✏ common/layers.py）。

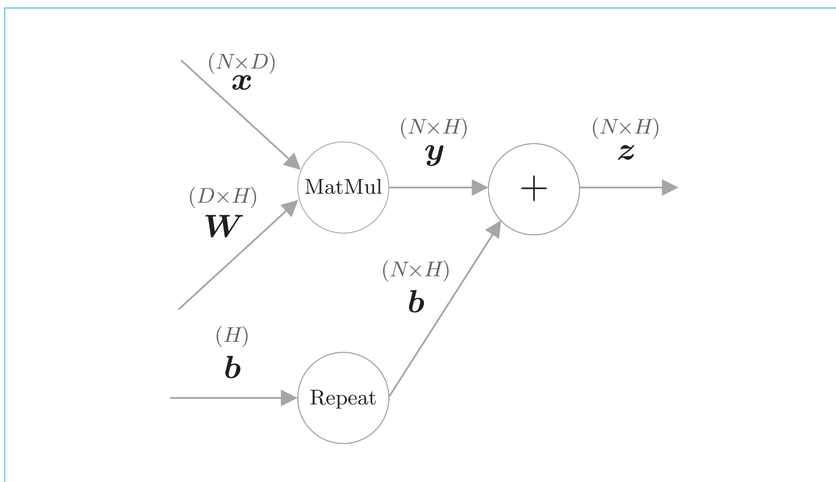


图 1-29　Affine 层的计算图

```
class Affine:
    def __init__(self, W, b):
        self.params = [W, b]
        self.grads = [np.zeros_like(W), np.zeros_like(b)]
        self.x = None

    def forward(self, x):
        W, b = self.params
        out = np.dot(x, W) + b
        self.x = x
        return out

    def backward(self, dout):
        W, b = self.params
        dx = np.dot(dout, W.T)
        dW = np.dot(self.x.T, dout)
        db = np.sum(dout, axis=0)

        self.grads[0][...] = dW
        self.grads[1][...] = db
        return dx
```

根据本书的代码规范，Affine 层将参数保存在实例变量 `params` 中，将梯度保存在实例变量 `grads` 中。它的反向传播可以通过执行 MatMul 节点和 Repeat 节点的反向传播来实现。Repeat 节点的反向传播可以通过 `np.sum()` 计算出来，此时注意矩阵的形状，就可以清楚地知道应该对哪个轴（`axis`）求和。最后，将权重参数的梯度设置给实例变量 `grads`。以上就是 Affine 层的实现。



使用已经实现的MatMul层，可以更轻松地实现Affine层。这里出于复习的目的，没有使用MatMul层，而是使用NumPy的方法进行了实现。

1.3.5.3 Softmax with Loss 层

我们将 Softmax 函数和交叉熵误差一起实现为 Softmax with Loss 层。此时，计算图如图 1-30 所示。

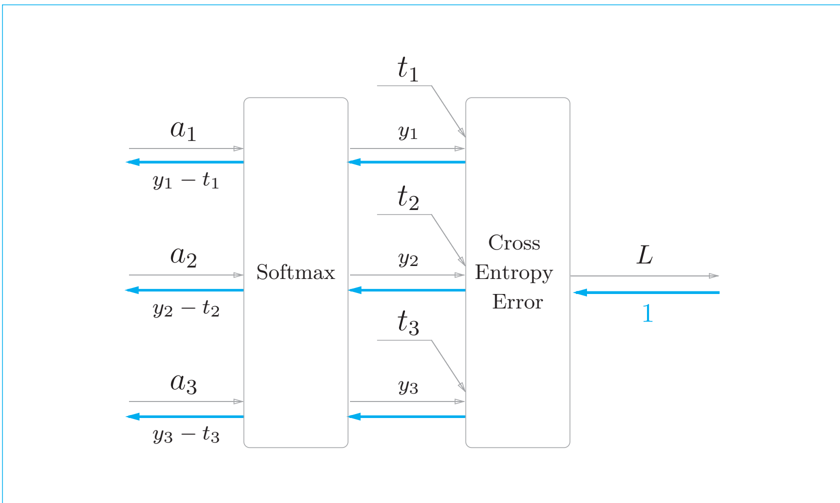


图1-30 Softmax with Loss层的计算图

图 1-30 的计算图将 Softmax 函数记为 Softmax 层，将交叉熵误差记为

Cross Entropy Error层。这里假设要执行3类别分类的任务，从前一层（靠近输入的层）接收 3 个输入。

如图 1-30 所示，Softmax 层对输入 a_1, a_2, a_3 进行正规化，输出 y_1, y_2, y_3。Cross Entropy Error 层接收 Softmax 的输出 y_1, y_2, y_3 和监督标签 t_1, t_2, t_3，并基于这些数据输出损失 L。



在图1-30中，需要注意的是反向传播的结果。从Softmax层传来的反向传播有 $y_1 - t_1, y_2 - t_2, y_3 - t_3$ 这样一个很“漂亮”的结果。因为 y_1, y_2, y_3 是Softmax层的输出，t_1, t_2, t_3 是监督标签，所以 $y_1 - t_1, y_2 - t_2, y_3 - t_3$ 是Softmax层的输出和监督标签的差分。神经网络的反向传播将这个差分（误差）传给前面的层。这是神经网络的学习中的一个重要性质。

这里我们省略对 Softmax with Loss 层的实现的说明，具体代码在 `common/layers.py` 中。另外，Softmax with Loss 层的反向传播的推导过程在前作《深度学习入门：基于 Python 的理论与实现》的附录 A 中有详细说明，感兴趣的读者可以参考一下。

1.3.6 权重的更新

通过误差反向传播法求出梯度后，就可以使用该梯度更新神经网络的参数。此时，神经网络的学习按如下步骤进行。

- 步骤 1：mini-batch

  从训练数据中随机选出多笔数据。

- 步骤 2：计算梯度

  基于误差反向传播法，计算损失函数关于各个权重参数的梯度。

- 步骤 3：更新参数

  使用梯度更新权重参数。

- 步骤 4：重复

  根据需要重复多次步骤 1、步骤 2 和步骤 3。

我们按照上面的步骤进行神经网络的学习。首先，选择 mini-batch 数据，根据误差反向传播法获得权重的梯度。这个梯度指向当前的权重参数所处位置中损失增加最多的方向。因此，通过将参数向该梯度的反方向更新，可以降低损失。这就是梯度下降法（gradient descent）。之后，根据需要将这一操作重复多次即可。

我们在上面的步骤 3 中更新权重。权重更新方法有很多，这里我们来实现其中最简单的随机梯度下降法（Stochastic Gradient Descent，SGD）。其中，“随机”是指使用随机选择的数据（mini-batch）的梯度。

SGD 是一个很简单的方法。它将（当前的）权重朝梯度的（反）方向更新一定距离。如果用数学式表示，则有：

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial L}{\partial \boldsymbol{W}} \tag{1.16}$$

这里将要更新的权重参数记为 $\boldsymbol{W}$，损失函数关于 $\boldsymbol{W}$ 的梯度记为 $\frac{\partial L}{\partial \boldsymbol{W}}$。$\eta$ 表示学习率，实际上使用 0.01、0.001 等预先定好的值。

现在，我们来进行 SGD 的实现。这里考虑到模块化，将进行参数更新的类实现在 `common/optimizer.py` 中。除了 SGD 之外，这个文件中还有 AdaGrad 和 Adam 等的实现。

进行参数更新的类的实现拥有通用方法 `update(params, grads)`。这里，在参数 `params` 和 `grads` 中分别以列表形式保存了神经网络的权重和梯度。此外，假定 `params` 和 `grads` 在相同索引处分别保存了对应的参数和梯度。这样一来，SGD 就可以像下面这样实现（✏ `common/optimizer.py`）。

```
class SGD:
    def __init__(self, lr=0.01):
        self.lr = lr

    def update(self, params, grads):
        for i in range(len(params)):
            params[i] -= self.lr * grads[i]
```

初始化参数 lr 表示学习率（learning rate）。这里将学习率保存为实例变量。然后，在 update(params, grads) 方法中实现参数的更新处理。

使用这个 SGD 类，神经网络的参数更新可按如下方式进行（下面的代码是不能实际运行的伪代码）。

```
model = TwoLayerNet(...)
optimizer = SGD()

for i in range(10000):
    ...
    x_batch, t_batch = get_mini_batch(...) # 获取mini-batch
    loss = model.forward(x_batch, t_batch)
    model.backward()
    optimizer.update(model.params, model.grads)
    ...
```

像这样，通过独立实现进行最优化的类，系统的模块化会变得更加容易。除了 SGD 外，本书还实现了 AdaGrad 和 Adam 等方法，它们的实现都在 common/optimizer.py 中。这里省略对这些最优化方法的介绍，详细内容请参考前作《深度学习入门：基于 Python 的理论与实现》的 6.1 节。

1.4　使用神经网络解决问题

到这里为止，我们的准备工作就做好了。现在，我们对一个简单的数据集进行神经网络的学习。

1.4.1　螺旋状数据集

本书在 dataset 目录中提供了几个便于处理数据集的类，本节将使用其中的 dataset/spiral.py 文件。这个文件中实现了读取螺旋（旋涡）状数据的类，其用法如下所示（✏ ch01/show_spiral_dataset.py）。

```
import sys
sys.path.append('..')  # 为了引入父目录的文件而进行的设定
```

```
from dataset import spiral
import matplotlib.pyplot as plt


x, t = spiral.load_data()
print('x', x.shape) # (300, 2)
print('t', t.shape) # (300, 3)
```

在上面的例子中，要从 ch01 目录的 dataset 目录引入 spiral.py。因此，上面的代码通过 sys.path.append('..') 将父目录添加到了 import 的检索路径中。

然后，使用 spiral.load_data() 进行数据的读入。此时，x 是输入数据，t 是监督标签。观察 x 和 t 的形状，可知它们各自有 300 笔样本数据，其中 x 是二维数据，t 是三维数据。另外，t 是 one-hot 向量，对应的正确解标签的类标记为 1，其余的标记为 0。下面，我们把这些数据绘制在图上，结果如图 1-31 所示。

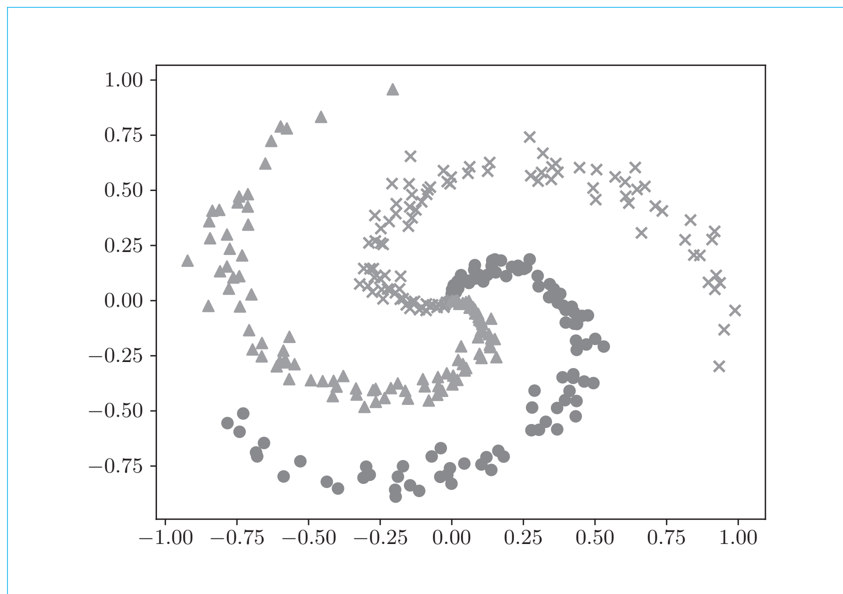


图1-31　学习用的螺旋状数据集（用×▲●分别表示3个类）

如图 1-31 所示，输入是二维数据，类别数是 3。观察这个数据集可知，它不能被直线分割。因此，我们需要学习非线性的分割线。那么，我们的神经网络（具有使用非线性的 sigmoid 激活函数的隐藏层的神经网络）能否正确学习这种非线性模式呢？让我们实验一下。



因为这个实验相对简单，所以我们不把数据集分成训练数据、验证数据和测试数据。不过，实际任务中会将数据集分为训练数据和测试数据（以及验证数据）来进行学习和评估。

1.4.2　神经网络的实现

现在，我们来实现一个具有一个隐藏层的神经网络。首先，import 语句和初始化程序的 __init__() 如下所示（✏ ch01/two_layer_net.py）。

```
import sys
sys.path.append('..')
import numpy as np
from common.layers import Affine, Sigmoid, SoftmaxWithLoss


class TwoLayerNet:
    def __init__(self, input_size, hidden_size, output_size):
        I, H, O = input_size, hidden_size, output_size

        # 初始化权重和偏置
        W1 = 0.01 * np.random.randn(I, H)
        b1 = np.zeros(H)
        W2 = 0.01 * np.random.randn(H, O)
        b2 = np.zeros(O)

        # 生成层
        self.layers = [
            Affine(W1, b1),
            Sigmoid(),
            Affine(W2, b2)
        ]
        self.loss_layer = SoftmaxWithLoss()
```

```
        # 将所有的权重和梯度整理到列表中
        self.params, self.grads = [], []
        for layer in self.layers:
            self.params += layer.params
            self.grads += layer.grads
```

初始化程序接收 3 个参数。input_size 是输入层的神经元数，hidden_size 是隐藏层的神经元数，output_size 是输出层的神经元数。在内部实现中，首先用零向量（np.zeros()）初始化偏置，再用小的随机数（0.01 * np.random.randn()）初始化权重。通过将权重设成小的随机数，学习可以更容易地进行。接着，生成必要的层，并将它们整理到实例变量 layers 列表中。最后，将这个模型使用到的参数和梯度归纳在一起。



因为Softmax with Loss层和其他层的处理方式不同，所以不将它放入layers列表中，而是单独存储在实例变量loss_layer中。

接着，我们为 TwoLayerNet 实现 3 个方法，即进行推理的 predict() 方法、正向传播的 forward() 方法和反向传播的 backward() 方法（✏ ch01/two_layer_net.py）。

```
    def predict(self, x):
        for layer in self.layers:
            x = layer.forward(x)
        return x

    def forward(self, x, t):
        score = self.predict(x)
        loss = self.loss_layer.forward(score, t)
        return loss

    def backward(self, dout=1):
        dout = self.loss_layer.backward(dout)
        for layer in reversed(self.layers):
            dout = layer.backward(dout)
        return dout
```

如上所示，这个实现非常清楚。因为我们已经将神经网络中要用的处理模块实现为了层，所以这里只需要以合理的顺序调用这些层的 forward() 方法和 backward() 方法即可。

1.4.3　学习用的代码

下面，我们来看一下学习用的代码。首先，读入学习数据，生成神经网络（模型）和优化器。然后，按照之前介绍的学习的 4 个步骤进行学习。另外，在机器学习领域，通常将针对具体问题设计的方法（神经网络、SVM 等）称为模型。学习用的代码如下所示（✏ ch01/train_custom_loop.py）。

```
import sys
sys.path.append('..')
import numpy as np
from common.optimizer import SGD
from dataset import spiral
import matplotlib.pyplot as plt
from two_layer_net import TwoLayerNet


# ❶设定超参数
max_epoch = 300
batch_size = 30
hidden_size = 10
learning_rate = 1.0

# ❷读入数据，生成模型和优化器
x, t = spiral.load_data()
model = TwoLayerNet(input_size=2, hidden_size=hidden_size, output_size=3)
optimizer = SGD(lr=learning_rate)

# 学习用的变量
data_size = len(x)
max_iters = data_size // batch_size
total_loss = 0
loss_count = 0
loss_list = []
```

```
for epoch in range(max_epoch):
    # ❸打乱数据
    idx = np.random.permutation(data_size)
    x = x[idx]
    t = t[idx]

    for iters in range(max_iters):
        batch_x = x[iters*batch_size:(iters+1)*batch_size]
        batch_t = t[iters*batch_size:(iters+1)*batch_size]

    # ❹计算梯度，更新参数
    loss = model.forward(batch_x, batch_t)
    model.backward()
    optimizer.update(model.params, model.grads)

    total_loss += loss
    loss_count += 1

    # ❺定期输出学习过程
    if (iters+1) % 10 == 0:
        avg_loss = total_loss / loss_count
        print('| epoch %d |  iter %d / %d | loss %.2f'
              % (epoch + 1, iters + 1, max_iters, avg_loss))
        loss_list.append(avg_loss)
        total_loss, loss_count = 0, 0
```

首先，在代码❶的地方设定超参数。具体而言，就是设定学习的 epoch 数、mini-batch 的大小、隐藏层的神经元数和学习率。接着，在代码❷的地方进行数据的读入，生成神经网络（模型）和优化器。我们已经将 2 层神经网络实现为了 `TwoLayerNet` 类，将优化器实现为了 `SGD` 类，这里直接使用它们就可以。



epoch 表示学习的单位。1 个 epoch 相当于模型"看过"一遍所有的学习数据（遍历数据集）。这里我们进行 300 个 epoch 的学习。

在进行学习时，需要随机选择数据作为 mini-batch。这里，我们以

epoch 为单位打乱数据，对于打乱后的数据，按顺序从头开始抽取数据。数据的打乱（准确地说，是数据索引的打乱）使用 np.random.permutation() 方法。给定参数 N，该方法可以返回 0 到 $N-1$ 的随机序列，其实际的使用示例如下所示。

```
>>> import numpy as np
>>> np.random.permutation(10)
array([7, 6, 8, 3, 5, 0, 4, 1, 9, 2])

>>> np.random.permutation(10)
array([1, 5, 7, 3, 9, 2, 8, 6, 0, 4])
```

像这样，调用 np.random.permutation() 可以随机打乱数据的索引。

接着，在代码❹的地方计算梯度，更新参数。最后，在代码❺的地方定期地输出学习结果。这里，每 10 次迭代计算 1 次平均损失，并将其添加到变量 loss_list 中。以上就是学习用的代码。



这里实现的神经网络的学习用的代码在本书其他地方也可以使用。因此，本书将这部分代码作为 Trainer 类提供出来。使用 Trainer 类，可以将神经网络的学习细节嵌入 Trainer 类。详细的用法将在 1.4.4 节说明。

运行一下上面的代码（ch01/train_custom_loop.py）就会发现，向终端输出的损失的值在平稳下降。我们将结果画出来，如图 1-32 所示。

由图 1-32 可知，随着学习的进行，损失在减小。我们的神经网络正在朝着正确的方向学习！接下来，我们将学习后的神经网络的区域划分（也称为决策边界）可视化，结果如图 1-33 所示。

由图 1-33 可知，学习后的神经网络可以正确地捕获“旋涡”这个模式。也就说，模型正确地学习了非线性的区域划分。像这样，神经网络通过隐藏层可以实现复杂的表现力。深度学习的特征之一就是叠加的层越多，表现力越丰富。

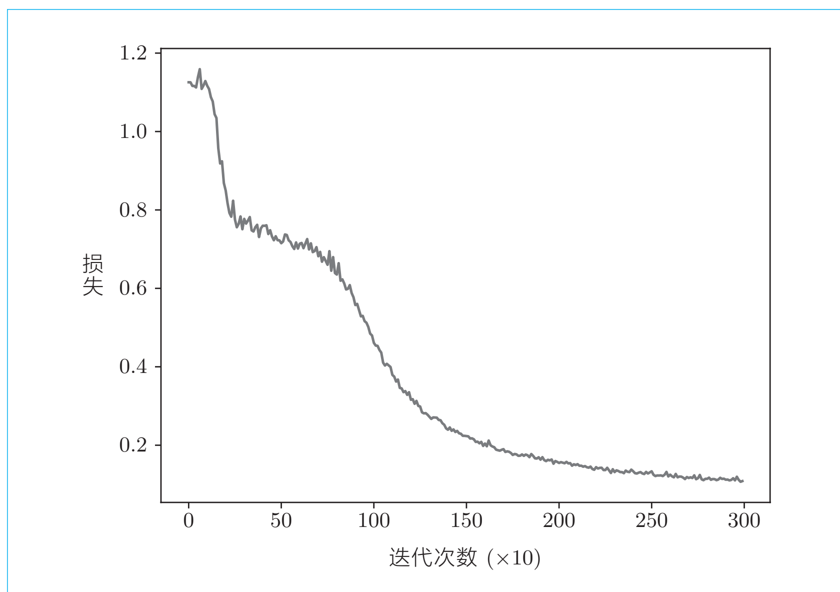


图1-32 损失的图形：横轴是学习的迭代次数（刻度值的10倍），竖轴是每10次迭代的平均损失



图1-33 学习后的神经网络的决策边界（用不同颜色描绘神经网络识别的各个类别的区域）

1.4.4　Trainer类

如前所述，本书中有很多机会执行神经网络的学习。为此，就需要编写前面那样的学习用的代码。然而，每次都写相同的代码太无聊了，因此我们将进行学习的类作为 Trainer 类提供出来。Trainer 类的内部实现和刚才的源代码几乎相同，只是添加了一些新的功能而已，我们在需要的时候再详细说明其用法。

Trainer 类的代码在 common/trainer.py 中。这个类的初始化程序接收神经网络（模型）和优化器，具体如下所示。

```
model = TwoLayerNet(...)
optimizer = SGD(lr=1.0)
trainer = Trainer(model, optimizer)
```

然后，调用 fit() 方法开始学习。fit() 方法的参数如表 1-1 所示。

表 1-1　Trainer类的 fit() 方法的参数。表中的 "(=XX)" 表示参数的默认值

参数	说明
x	输入数据
t	监督标签
max_epoch（= 10）	进行学习的 epoch 数
batch_size（= 32）	mini-batch 的大小
eval_interval（= 20）	输出结果（平均损失等）的间隔。 例如设置 eval_interval=20，则每 20 次迭代计算 1 次平均损失，并将结果输出到界面上
max_grad（= None）	梯度的最大范数。 当梯度的范数超过这个值时，缩小梯度（梯度裁剪，具体请参考第 5 章）

另外，Trainer 类有 plot() 方法，它将 fit() 方法记录的损失（准确地说，是按照 eval_interval 评价的平均损失）在图上画出来。使用 Trainer 类进行学习的代码如下所示（✏ ch01/train.py）。

```
import sys
sys.path.append('..')
from common.optimizer import SGD
from common.trainer import Trainer
from dataset import spiral
from two_layer_net import TwoLayerNet


max_epoch = 300
batch_size = 30
hidden_size = 10
learning_rate = 1.0

x, t = spiral.load_data()
model = TwoLayerNet(input_size=2, hidden_size=hidden_size, output_size=3)
optimizer = SGD(lr=learning_rate)

trainer = Trainer(model, optimizer)
trainer.fit(x, t, max_epoch, batch_size, eval_interval=10)
trainer.plot()
```

执行这段代码，会进行和之前一样的神经网络的学习。通过将之前展示的学习用的代码交给 `Trainer` 类负责，代码变简洁了。本书今后都将使用 `Trainer` 类进行学习。

1.5 计算的高速化

神经网络的学习和推理需要大量的计算。因此，如何高速地计算神经网络是一个重要课题。本节将简单介绍一下可以有效加速神经网络的计算的位精度和 GPU 的相关内容。



相比计算的高速化，本书更加重视实现的易理解性。但是，从计算的高速化的角度出发，之后进行的实现将考虑数据的位精度。另外，在需要花费大量时间进行计算的地方，会将代码设计为可在 GPU 上执行。

1.5.1　位精度

NumPy 的浮点数默认使用 64 位的数据类型。不过，是否为 64 位还依赖于具体的环境，包括操作系统、Python 和 NumPy 的版本等。我们可以使用下面的代码来验证是否使用了 64 位浮点数。

```
>>> import numpy as np
>>> a = np.random.randn(3)
>>> a.dtype
dtype('float64')
```

通过 NumPy 数组的实例变量 `dtype`，可以查看数据类型。上面的结果是 `float64`，表示 64 位的浮点数。

NumPy 中默认使用 64 位浮点数。但是，我们已经知道使用 32 位浮点数也可以无损地（识别精度几乎不下降）进行神经网络的推理和学习。从内存的角度来看，因为 32 位只有 64 位的一半，所以通常首选 32 位。另外，在神经网络的计算中，数据传输的总线带宽有时会成为瓶颈。在这种情况下，毫无疑问数据类型也是越小越好。再者，就计算速度而言，32 位浮点数也能更高速地进行计算（浮点数的计算速度依赖于 CPU 或 GPU 的架构）。

因此，本书优先使用 32 位浮点数。要在 NumPy 中使用 32 位浮点数，可以像下面这样将数据类型指定为 `np.float32` 或者 `'f'`。

```
>>> b = np.random.randn(3).astype(np.float32)
>>> b.dtype
dtype('float32')

>>> c = np.random.randn(3).astype('f')
>>> c.dtype
dtype('float32')
```

另外，我们已经知道，如果只是神经网络的推理，则即使使用 16 位浮点数进行计算，精度也基本上不会下降 [6]。不过，虽然 NumPy 中准备有

16 位浮点数，但是普通 CPU 或 GPU 中的运算是用 32 位执行的。因此，即便变换为 16 位浮点数，因为计算本身还是用 32 位浮点数执行的，所以处理速度方面并不能获得什么好处。

但是，如果是要（在外部文件中）保存学习好的权重，则 16 位浮点数是有用的。具体地说，将权重数据用 16 位精度保存时，只需要 32 位时的一半容量。因此，本书仅在保存学习好的权重时，将其变换为 16 位浮点数。



随着深度学习备受瞩目，最近的GPU已经开始支持16位半精度浮点数的存储与计算。另外，谷歌公司设计了一款名为TPU的专用芯片，可以支持8位计算[7]。

1.5.2 GPU（CuPy）

深度学习的计算由大量的乘法累加运算组成。这些乘法累加运算的绝大部分可以并行计算，这是 GPU 比 CPU 擅长的地方。因此，一般的深度学习框架都被设计为既可以在 CPU 上运行，也可以在 GPU 上运行。

本书中可以选用 Python 库 CuPy[3]。CuPy 是基于 GPU 进行并行计算的库。要使用 CuPy，需要使用安装有 NVIDIA 的 GPU 的机器，并且需要安装 CUDA 这个面向 GPU 的通用并行计算平台。详细的安装方法请参考 CuPy 的官方安装文档[4]。

使用 Cupy，可以轻松地使用 NVIDIA 的 GPU 进行并行计算。更重要的是，CuPy 和 NumPy 拥有共同的 API。下面我们来看一个简单的使用示例。

```
>>> import cupy as cp
>>> x = cp.arange(6).reshape(2, 3).astype('f')
>>> x
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]], dtype=float32)
>>> x.sum(axis=1)
array([  3.,  12.], dtype=float32)
```

如上所示，CuPy 的使用方法与 NumPy 基本相同。另外，它的内部使用 GPU 进行计算。这意味着使用 NumPy 写的代码可以轻松地改成“GPU 版”，因为我们要做的（基本上）只是把 numpy 替换为 cupy 而已。



截至2018年6月，CuPy并没有完全覆盖NumPy的方法。虽然CuPy和NumPy并不完全兼容，但是它们有许多共同的API。

重申一下，本书为了使代码实现易于理解，基本上都基于 CPU 进行实现。对于计算上要耗费大量时间的代码，则提供使用了 CuPy 的实现（可选）。不过，即便在使用 CuPy 的情况下，也会尽量做到不让读者感到是在使用 CuPy。

本书中可以在 GPU 上运行的代码最先出现在第 4 章（ch04/train.py）。这个 ch04/train.py 从以下的 import 语句开始。

```
import sys
sys.path.append('..')
import numpy as np
from common import config
# 在用GPU运行时，请打开下面的注释（需要cupy）
# ===============================================
# config.GPU = True
# ===============================================
...
```

用 CPU 执行上述代码需要花费几个小时，但是如果使用 GPU，则只需要几十分钟。并且，只要修改上述源代码中的一行，本书提供的代码就可以在 GPU 模式下运行。具体而言，只需打开注释 # config.GPU = True，并用 CuPy 代替 NumPy 即可。如此，代码就可以在 GPU 上运行，并高速地进行学习。请有 GPU 的读者多多尝试一下。



将NumPy切换为CuPy的机制非常简单，感兴趣的读者请参考 common/config.py、common/np.py 和 common/layers.py 的 import 语句。

1.6　小结

本章我们复习了神经网络的基础。首先回顾了向量和矩阵等数学知识，确认了 Python（特别是 NumPy）的基本用法。然后，我们观察了神经网络的结构。另外，我们还讨论了几个计算图的基本部件（加法节点、乘法节点等），并介绍了它们的正向传播和反向传播。

接着，我们进行了神经网络的实现。考虑到模块化，我们将神经网络的基本部件实现为了层。在层的实现中，我们制定了本书的代码规范，即具有类方法 `forward()` 和 `backward()`，以及实例变量 `params` 和 `grads`。这使得神经网络的实现更加清晰。

最后，我们对人造的螺旋状数据集使用具有一个隐藏层的神经网络进行了学习，并确认了模型学习的正确性。到这里为止，神经网络的复习就结束了。现在，让我们手握神经网络这一可靠的武器，迈入自然语言处理的世界。出发吧！

本章所学的内容

- 神经网络具有输入层、隐藏层和输出层
- 通过全连接层进行线性变换，通过激活函数进行非线性变换
- 全连接层和 mini-batch 处理都可以写成矩阵计算
- 使用误差反向传播法可以高效地求解神经网络的损失的梯度
- 使用计算图能够将神经网络中发生的处理可视化，这有助于理解正向传播和反向传播
- 在神经网络的实现中，通过将组件模块化为层，可以简化实现
- 数据的位精度和 GPU 并行计算对神经网络的高速化非常重要

第2章 自然语言和单词的分布式表示

Marty: "This is heavy（棘手）."

Dr. Brown: "In the future, things are so heavy（重）?"

——电影《回到未来》

接下来，我们将踏入自然语言处理的世界。自然语言处理涉及多个子领域，但是它们的根本任务都是让计算机理解我们的语言。何谓让计算机理解我们的语言？存在哪些方法？本章我们将以这些问题为中心展开讨论。为此，我们将先详细考察古典方法，即深度学习出现以前的方法。从下一章开始，再介绍基于深度学习（确切地说，是神经网络）的方法。

本章我们还会练习使用Python处理文本，实现分词（将文本分割成单词）和单词ID化（将单词转换为单词ID）等任务。本章实现的函数在后面的章节中也会用到。因此，本章也可以说是后续文本处理的准备工作。那么，让我们一起进入自然语言处理的世界吧！

2.1 什么是自然语言处理

我们平常使用的语言，如日语或英语，称为自然语言（natural language）。所谓自然语言处理（Natural Language Processing，NLP），顾名思义，就是处理自然语言的科学。简单地说，它是一种能够让计算机理解人类语言的技术。换言之，自然语言处理的目标就是让计算机理解人说的话，进而完成对我们有帮助的事情。

另外，说到计算机可以理解的语言，我们可能会想到编程语言或者标记语言等。这些语言的语法定义可以唯一性地解释代码含义，计算机也能根据确定的规则分析代码。

众所周知，编程语言是一种机械的、缺乏活力的语言。换句话说，它是一种“硬语言”。而英语或日语等自然语言是“软语言”。这里的“软”是指意思和形式会灵活变化，比如含义相同的文章可以有不同的表述，或者文章存在歧义，等等。另外，自然语言的“软”还体现在，新的词语或新的含义会随着时代的发展不断出现。

自然语言是活着的语言，具有“柔软性”。因此，要让“头脑僵硬”的计算机去理解自然语言，使用常规方法是无法办到的。但是，如果我们能办到，就能让计算机去完成一些对人们有用的事情。事实上，我们可以看到很多这样的应用。搜索引擎和机器翻译就是两个比较好理解的例子，除此之外，还有问答系统、假名汉字转化（IME）、自动文本摘要和情感分析等，在我们身边已经使用了很多自然语言处理技术。



问答系统是自然语言处理技术的一个应用。作为代表，国际商业机器公司（IBM）的Watson系统非常有名。2011年，美国的一档答题闯关节目《危险边缘》（*Jeopardy!*）让Watson名声大噪。在这个节目中，Watson的答题比任何人都准确，战胜了往届的冠军（不过，给Watson的问题是用文本给出的）。这一“事件”引起了世人的广泛关注，大概也是从这个时候开始，人们对人工智能的期待和不安都开始增加。此外，IBM将Watson称为决策支援系统。最近，有媒体报道，Watson利用过往的大量医疗数据，针对疑难症提供了正确的治疗建议，挽救了患者生命。

单词含义

我们的语言是由文字构成的，而语言的含义是由单词构成的。换句话说，单词是含义的最小单位。因此，为了让计算机理解自然语言，让它理解单词含义可以说是最重要的事情了。

本章的主题是让计算机理解单词含义。确切地说，我们将探讨一些巧妙地蕴含了单词含义的表示方法。具体来说，本章和下一章将讨论以下 3 种方法。

- 基于同义词词典的方法 本章
- 基于计数的方法 本章
- 基于推理的方法（word2vec） 下一章

首先，我们将简单介绍一下使用人工整理好的同义词词典的方法。然后，对利用统计信息表示单词的方法（这里称为“基于计数的方法”）进行说明。这些都是本章学习的内容。在下一章，我们将讨论利用神经网络的基于推理的方法（具体来说，就是 word2vec 方法）。本章的结构参考了斯坦福大学课程“CS224d: Deep Learning for Natural Language Processing”[10]。

2.2 同义词词典

要表示单词含义，首先可以考虑通过人工方式来定义单词含义。一种方法是像《新华字典》那样，一个词一个词地说明单词含义。比如，当你用字典查“汽车”这个单词时，就会看到“装有车轮并依靠它们前行的交通工具或运输工具……”这样的说明。通过像这样定义单词，计算机或许也能够理解单词含义。

回顾自然语言处理的历史，人们已经尝试过很多次类似这样的人工定义单词含义的活动。但是，目前被广泛使用的并不是《新华字典》那样的常规词典，而是一种被称为同义词词典（thesaurus）的词典。在同义词词典中，具有相同含义的单词（同义词）或含义类似的单词（近义词）被归类到同一个组中。比如，使用同义词词典，我们可以知道 car 的同义词有 automobile、motorcar 等（图 2-1）。



图2-1 同义词的例子：car、auto和automobile等都是表示“汽车”的同义词

另外，在自然语言处理中用到的同义词词典有时会定义单词之间的粒度更细的关系，比如“上位 – 下位”关系、“整体 – 部分”关系。举个例子，如图 2-2 所示，我们利用图结构定义了各个单词之间的关系。

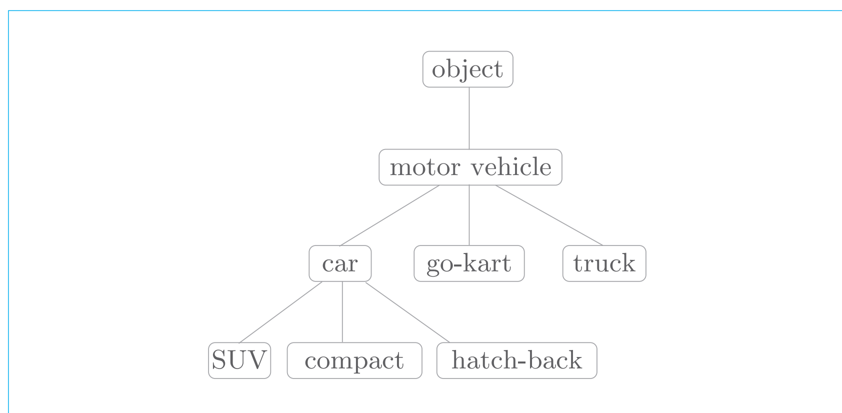


图2-2 根据各单词的含义，基于上位 – 下位关系形成的图（参考文献[14]）

在图 2-2 中，单词 motor vehicle（机动车）是单词 car 的上位概念。car 的下位概念有 SUV、compact 和 hatch-back 等更加具体的车种。

像这样，通过对所有单词创建近义词集合，并用图表示各个单词的关系，可以定义单词之间的联系。利用这个“单词网络”，可以教会计算机单词之间的相关性。也就是说，我们可以将单词含义（间接地）教给计算机，然后利用这一知识，就能让计算机做一些对我们有用的事情。



如何使用同义词词典根据自然语言处理的具体应用的不同而不同。比如，在信息检索场景中，如果事先知道automobile和car是近义词，就可以将automobile的检索结果添加到car的检索结果中。

2.2.1 WordNet

在自然语言处理领域，最著名的同义词词典是 WordNet[17]。WordNet 是普林斯顿大学于 1985 年开始开发的同义词词典，迄今已用于许多研究，并活跃于各种自然语言处理应用中。

使用 WordNet，可以获得单词的近义词，或者利用单词网络。使用单词网络，可以计算单词之间的相似度。这里，我们不对 WordNet 进行详细说明，对 WordNet 的 Python 实现感兴趣的读者，可以参考附录 B。在附录 B 中，我们会安装 WordNet（准确地说，是安装 NLTK 模块），并进行一些简单的实验。



在附录 B 中，我们将实际使用 WordNet 来计算单词之间的相似度。具体来说，就是基于一个人工定义的单词网络，来计算单词之间的相似度。如果能（在一定程度上正确）计算单词之间的相似度，那么我们就踏出了理解单词含义的第一步。

2.2.2 同义词词典的问题

WordNet 等同义词词典中对大量单词定义了同义词和层级结构关系等。利用这些知识，可以（间接地）让计算机理解单词含义。不过，人工标记也存在一些较大的缺陷。下面，我们就来看一下同义词词典的主要问题，并分别对其进行简要说明。

难以顺应时代变化

我们使用的语言是活的。随着时间的推移，新词不断出现，而那些落满尘埃的旧词不知哪天就会被遗忘。比如，“众筹”（crowdfunding）就是一个最近才开始使用的新词。

另外，语言的含义也会随着时间的推移而变化。比如，英语中的 heavy 一词，现在有“事态严重”的含义（主要用作俚语），但以前是

没有这种用法的。在电影《回到未来》中，有这样一个场景：从 1985 年穿越回来的马蒂和生活在 1955 年的博士的对话中，对 heavy 的含义有不同的理解。如果要处理这样的单词变化，就需要人工不停地更新同义词词典。

人力成本高

制作词典需要巨大的人力成本。以英文为例，据说现有的英文单词总数超过 1000 万个。在极端情况下，还需要对如此大规模的单词进行单词之间的关联。顺便提一下，WordNet 中收录了超过 20 万个的单词。

无法表示单词的微妙差异

同义词词典中将含义相近的单词作为近义词分到一组。但实际上，即使是含义相近的单词，也有细微的差别。比如，vintage 和 retro 虽然表示相同的含义，但是用法不同，而这种细微的差别在同义词词典中是无法表示出来的（让人来解释是相当困难的）。

因此，使用同义词词典，即人工定义单词含义的方法存在很多问题。为了避免这些问题，接下来我们将介绍基于计数的方法和利用神经网络的基于推理的方法。这两种方法可以从海量的文本数据中自动提取单词含义，将我们从人工关联单词的辛苦劳动中解放出来。



不仅限于自然语言处理，在图像识别领域，多年来也一直是人工设计特征量。但是，随着深度学习的出现，现在从原始图像直接获得最终结果已成为可能，人为介入的必要性大幅降低。在自然语言处理领域也有类似现象。也就是说，我们正在从人工制作词典或设计特征量的旧范式，向尽量减少人为干预的、仅从文本数据中获取最终结果的新范式转移。

2.3 基于计数的方法

从介绍基于计数的方法开始，我们将使用语料库（corpus）。简而言之，语料库就是大量的文本数据。不过，语料库并不是胡乱收集数据，一般收集的都是用于自然语言处理研究和应用的文本数据。

说到底，语料库只是一些文本数据而已。不过，其中的文章都是由人写出来的。换句话说，语料库中包含了大量的关于自然语言的实践知识，即文章的写作方法、单词的选择方法和单词含义等。基于计数的方法的目标就是从这些富有实践知识的语料库中，自动且高效地提取本质。



自然语言处理领域中使用的语料库有时会给文本数据添加额外的信息。比如，可以给文本数据的各个单词标记词性。在这种情况下，为了方便计算机处理，语料库通常会被结构化（比如，采用树结构等数据形式）。这里，假定我们使用的语料库没有添加标签，而是作为一个大的文本文件，只包含简单的文本数据。

2.3.1 基于Python的语料库的预处理

自然语言处理领域存在各种各样的语料库。说到有名的语料库，有 Wikipedia 和 Google News 等。另外，莎士比亚、夏目漱石等伟大作家的作品集也会被用作语料库。本章我们先使用仅包含一个句子的简单文本作为语料库，然后再处理更实用的语料库。

现在，我们使用 Python 的交互模式，对一个非常小的文本数据（语料库）进行预处理。这里所说的预处理是指，将文本分割为单词（分词），并将分割后的单词列表转化为单词 ID 列表。

下面，我们一边确认一边实现。首先来看一下作为语料库的样本文章。

```
>>> text = 'You say goodbye and I say hello.'
```

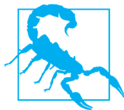
这里我们使用由单个句子构成的文本作为语料库。本来文本（text）应

该包含成千上万个（连续的）句子，但是，考虑到简洁性，这里先对这个小的文本数据进行预处理。下面，我们对上面的 text 进行分词。

```
>>> text = text.lower()
>>> text = text.replace('.', ' .')
>>> text
'you say goodbye and i say hello .'

>>> words = text.split(' ')
>>> words
['you', 'say', 'goodbye', 'and', 'i', 'say', 'hello', '.']
```

首先，使用 lower() 方法将所有字母转化为小写，这样可以将句子开头的单词也作为常规单词处理。然后，将空格作为分隔符，通过 split(' ') 切分句子。考虑到句子结尾处的句号（.），我们先在句号前插入一个空格（即用“ .”替换“.”），再进行分词。



这里，在进行分词时，我们采用了一种在句号前插入空格的“临时对策”，其实还有更加聪明、更加通用的实现方式，比如使用正则表达式。通过导入正则表达式的 re 模块，使用 re.split('(\W+)?', text) 也可以进行分词。关于正则表达式的详细信息，可以参考文献 [15]。

现在，我们已经可以将原始文章作为单词列表使用了。虽然分词后文本更容易处理了，但是直接以文本的形式操作单词，总感觉有些不方便。因此，我们进一步给单词标上 ID，以便使用单词 ID 列表。为此，我们使用 Python 的字典来创建单词 ID 和单词的对应表。

```
>>> word_to_id = {}
>>> id_to_word = {}
>>>
>>> for word in words:
...     if word not in word_to_id:
...         new_id = len(word_to_id)
...         word_to_id[word] = new_id
...         id_to_word[new_id] = word
```

变量 id_to_word 负责将单词 ID 转化为单词（键是单词 ID，值是单词），word_to_id 负责将单词转化为单词 ID。这里，我们从头开始逐一观察分词后的 words 的各个元素，如果单词不在 word_to_id 中，则分别向 word_to_id 和 id_to_word 添加新 ID 和单词。另外，我们将字典的长度设为新的单词 ID，单词 ID 按 $0, 1, 2, \cdots$ 逐渐增加。

这样一来，我们就创建好了单词 ID 和单词的对应表。下面，我们来实际看一下它们的内容。

```
>>> id_to_word
{0: 'you', 1: 'say', 2: 'goodbye', 3: 'and', 4: 'i', 5: 'hello', 6:'.'}
>>> word_to_id
{'you': 0, 'say': 1, 'goodbye': 2, 'and': 3, 'i': 4, 'hello': 5, '.': 6}
```

使用这些词典，可以根据单词检索单词 ID，或者反过来根据单词 ID 检索单词。我们实际尝试一下，如下所示。

```
>>> id_to_word[1]
'say'
>>> word_to_id['hello']
5
```

最后，我们将单词列表转化为单词 ID 列表。这里，我们使用 Python 的列表解析式将单词列表转化为单词 ID 列表，然后再将其转化为 NumPy 数组。

```
>>> import numpy as np
>>> corpus = [word_to_id[w] for w in words]
>>> corpus = np.array(corpus)
>>> corpus
array([0, 1, 2, 3, 4, 1, 5, 6])
```



列表解析式（list comprehension）或字典解析式（dict comprehension）是一种便于对列表或字典进行循环处理的写法。比如，要创建元素为列表 xs = [1,2,3,4] 中各个元素的平方的新列表，可以写成 [x**2 for x in xs]。

至此，我们就完成了利用语料库的准备工作。现在，我们将上述一系列处理实现为 preprocess() 函数（✏ common/util.py）。

```
def preprocess(text):
    text = text.lower()
    text = text.replace('.', ' .')
    words = text.split(' ')

    word_to_id = {}
    id_to_word = {}
    for word in words:
        if word not in word_to_id:
            new_id = len(word_to_id)
            word_to_id[word] = new_id
            id_to_word[new_id] = word

    corpus = np.array([word_to_id[w] for w in words])

    return corpus, word_to_id, id_to_word
```

使用这个函数，可以按如下方式对语料库进行预处理。

```
>>> text = 'You say goodbye and I say hello.'
>>> corpus, word_to_id, id_to_word = preprocess(text)
```

到这里，语料库的预处理就结束了。这里准备的 corpus、word_to_id 和 id_to_word 这 3 个变量名在本书接下来的很多地方都会用到。corpus 是单词 ID 列表，word_to_id 是单词到单词 ID 的字典，id_to_word 是单词 ID 到单词的字典。

现在，我们已经做好了操作语料库的准备，接下来的目标就是使用语料库提取单词含义。为此，本节我们将考察基于计数的方法。采用这种方法，我们能够将单词表示为向量。

2.3.2 单词的分布式表示

世界上存在各种各样的颜色，有的颜色被赋予了固定的名字，比如钴蓝

（cobalt blue）或者锌红（zinc red）；颜色也可以通过 RGB（Red/Green/Blue）三原色分别存在多少来表示。前者为不同的颜色赋予不同的名字，有多少种颜色，就需要有多少个不同的名字；后者则将颜色表示为三维向量。

需要注意的是，使用 RGB 这样的向量表示可以更准确地指定颜色，并且这种基于三原色的表示方式很紧凑，也更容易让人想象到具体是什么颜色。比如，即便不知道“深绯”是什么样的颜色，但如果知道它的 $(R, G, B) = (201, 23, 30)$，就至少可以知道它是红色系的颜色。此外，颜色之间的关联性（是否是相似的颜色）也更容易通过向量表示来判断和量化。

那么，能不能将类似于颜色的向量表示方法运用到单词上呢？更准确地说，可否在单词领域构建紧凑合理的向量表示呢？接下来，我们将关注能准确把握单词含义的向量表示。在自然语言处理领域，这称为分布式表示。



单词的分布式表示将单词表示为固定长度的向量。这种向量的特征在于它是用密集向量表示的。密集向量的意思是，向量的各个元素（大多数）是由非 0 实数表示的。例如，三维分布式表示是 `[0.21,-0.45,0.83]`。如何构建这样的单词的分布式表示是我们接下来的一个重要课题。

2.3.3　分布式假设

在自然语言处理的历史中，用向量表示单词的研究有很多。如果仔细看一下这些研究，就会发现几乎所有的重要方法都基于一个简单的想法，这个想法就是“某个单词的含义由它周围的单词形成”，称为分布式假设（distributional hypothesis）。许多用向量表示单词的近期研究也基于该假设。

分布式假设所表达的理念非常简单。单词本身没有含义，单词含义由它所在的上下文（语境）形成。的确，含义相同的单词经常出现在相同的语境中。比如“I drink beer.”“We drink wine.”，drink 的附近常有饮料出现。另外，从“I guzzle beer.”“We guzzle wine.”可知，guzzle 和 drink 所在的语境相似。进而我们可以推测出，guzzle 和 drink 是近义词（顺便说一下，

guzzle 是“大口喝”的意思）。

从现在开始，我们会经常使用“上下文”一词。本章说的上下文是指某个单词（关注词）周围的单词。在图 2-3 的例子中，左侧和右侧的 2 个单词就是上下文。

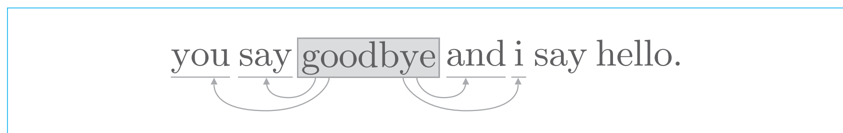


图 2-3　窗口大小为 2 的上下文例子。在关注 goodbye 时，将其左右各 2 个单词用作上下文

如图 2-3 所示，上下文是指某个居中单词的周围词汇。这里，我们将上下文的大小（即周围的单词有多少个）称为窗口大小（window size）。窗口大小为 1，上下文包含左右各 1 个单词；窗口大小为 2，上下文包含左右各 2 个单词，以此类推。



这里，我们将左右两边相同数量的单词作为上下文。但是，根据具体情况，也可以仅将左边的单词或者右边的单词作为上下文。此外，也可以使用考虑了句子分隔符的上下文。简单起见，本书仅处理不考虑句子分隔符、左右单词数量相同的上下文。

2.3.4　共现矩阵

下面，我们来考虑如何基于分布式假设使用向量表示单词，最直截了当的实现方法是对周围单词的数量进行计数。具体来说，在关注某个单词的情况下，对它的周围出现了多少次什么单词进行计数，然后再汇总。这里，我们将这种做法称为“基于计数的方法”，在有的文献中也称为“基于统计的方法”。

现在，我们就来看一下基于计数的方法。这里我们使用 2.3.1 节的语料库和 preprocess() 函数，再次进行预处理。

```
import sys
sys.path.append('..')
import numpy as np
```

```
from common.util import preprocess

text = 'You say goodbye and I say hello.'
corpus, word_to_id, id_to_word = preprocess(text)

print(corpus)
# [0 1 2 3 4 1 5 6]

print(id_to_word)
# {0: 'you', 1: 'say', 2: 'goodbye', 3: 'and', 4: 'i', 5: 'hello', 6:
 ↪  '.'}
```

从上面的结果可以看出，词汇总数为 7 个。下面，我们计算每个单词的上下文所包含的单词的频数。在这个例子中，我们将窗口大小设为 1，从单词 ID 为 0 的 you 开始。

从图 2-4 可以清楚地看到，单词 you 的上下文仅有 say 这个单词。用表格表示的话，如图 2-5 所示。

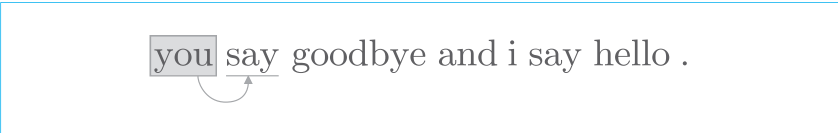


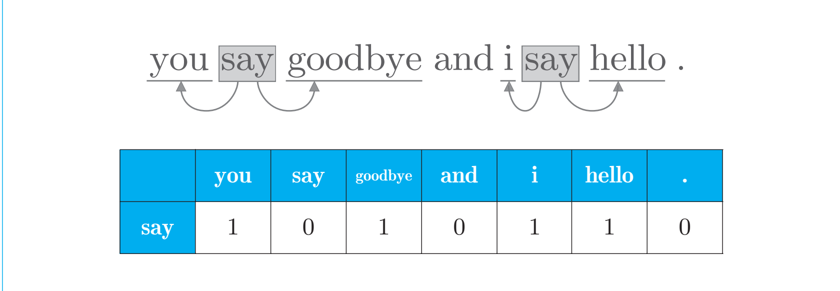
图 2-4　单词 you 的上下文

	you	say	goodbye	and	i	hello	.
you	0	1	0	0	0	0	0

图 2-5　用表格表示单词 you 的上下文中包含的单词的频数

图 2-5 表示的是作为单词 you 的上下文共现的单词的频数。同时，这也意味着可以用向量 `[0, 1, 0, 0, 0, 0, 0]` 表示单词 you。

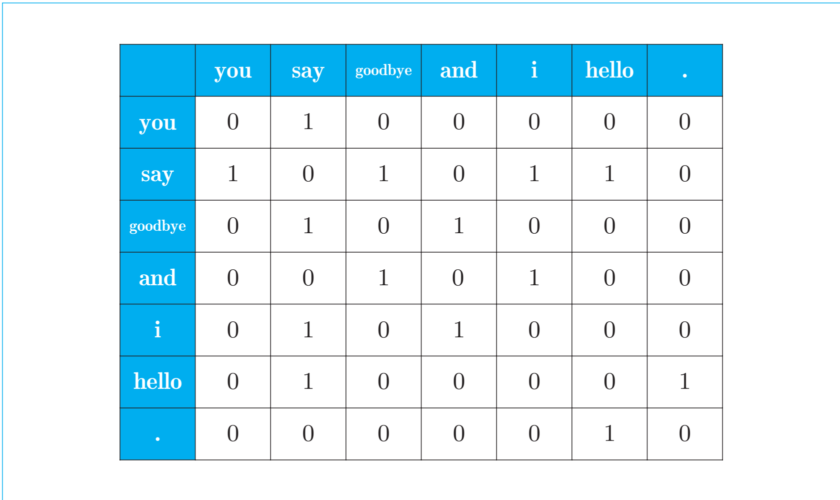
接着对单词 ID 为 1 的 say 进行同样的处理，结果如图 2-6 所示。



	you	say	goodbye	and	i	hello	.
say	1	0	1	0	1	1	0

图 2-6　用表格表示单词 say 的上下文中包含的单词的频数

从上面的结果可知，单词 say 可以表示为向量 `[1, 0, 1, 0, 1, 1, 0]`。对所有的 7 个单词进行上述操作，会得到如图 2-7 所示的结果。



	you	say	goodbye	and	i	hello	.
you	0	1	0	0	0	0	0
say	1	0	1	0	1	1	0
goodbye	0	1	0	1	0	0	0
and	0	0	1	0	1	0	0
i	0	1	0	1	0	0	0
hello	0	1	0	0	0	0	1
.	0	0	0	0	0	1	0

图 2-7　用表格汇总各个单词的上下文中包含的单词的频数

图 2-7 是汇总了所有单词的共现单词的表格。这个表格的各行对应相应单词的向量。因为图 2-7 的表格呈矩阵状，所以称为共现矩阵（co-occurence matrix）。

接下来，我们来实际创建一下上面的共现矩阵。这里，将图 2-7 的结果按原样手动输入。

```
C = np.array([
```

```
    [0, 1, 0, 0, 0, 0, 0],
    [1, 0, 1, 0, 1, 1, 0],
    [0, 1, 0, 1, 0, 0, 0],
    [0, 0, 1, 0, 1, 0, 0],
    [0, 1, 0, 1, 0, 0, 0],
    [0, 1, 0, 0, 0, 0, 1],
    [0, 0, 0, 0, 0, 1, 0],
], dtype=np.int32)
```

这就是共现矩阵。使用这个共现矩阵，可以获得各个单词的向量，如下所示。

```
print(C[0]) # 单词ID为0的向量
# [0 1 0 0 0 0 0]

print(C[4]) # 单词ID为4的向量
# [0 1 0 1 0 0 0]

print(C[word_to_id['goodbye']]) # goodbye的向量
# [0 1 0 1 0 0 0]
```

至此，我们通过共现矩阵成功地用向量表示了单词。上面我们是手动输入共现矩阵的，但这一操作显然可以自动化。下面，我们来实现一个能直接从语料库生成共现矩阵的函数。我们把这个函数称为 create_co_matrix(corpus, vocab_size, window_size=1)，其中参数 corpus 是单词 ID 列表，参数 vocab_size 是词汇个数，window_size 是窗口大小（✏ common/util.py）。

```
def create_co_matrix(corpus, vocab_size, window_size=1):
    corpus_size = len(corpus)
    co_matrix = np.zeros((vocab_size, vocab_size), dtype=np.int32)

    for idx, word_id in enumerate(corpus):
        for i in range(1, window_size + 1):
            left_idx = idx - i
            right_idx = idx + i

            if left_idx >= 0:
                left_word_id = corpus[left_idx]
                co_matrix[word_id, left_word_id] += 1

            if right_idx < corpus_size:
```

```
                right_word_id = corpus[right_idx]
                co_matrix[word_id, right_word_id] += 1

    return co_matrix
```

首先，用元素为 0 的二维数组对 co_matrix 进行初始化。然后，针对语料库中的每一个单词，计算它的窗口中包含的单词。同时，检查窗口内的单词是否超出了语料库的左端和右端。

这样一来，无论语料库多大，都可以自动生成共现矩阵。之后，我们都将使用这个函数生成共现矩阵。

2.3.5　向量间的相似度

前面我们通过共现矩阵将单词表示为了向量。下面，我们看一下如何测量向量间的相似度。

测量向量间的相似度有很多方法，其中具有代表性的方法有向量内积或欧式距离等。虽然除此之外还有很多方法，但是在测量单词的向量表示的相似度方面，余弦相似度（cosine similarity）是很常用的。设有 $\boldsymbol{x} = (x_1, x_2, x_3, \cdots, x_n)$ 和 $\boldsymbol{y} = (y_1, y_2, y_3, \cdots, y_n)$ 两个向量，它们之间的余弦相似度的定义如下式所示。

$$\text{similarity}(\boldsymbol{x}, \boldsymbol{y}) = \frac{\boldsymbol{x} \cdot \boldsymbol{y}}{\|\boldsymbol{x}\| \, \|\boldsymbol{y}\|} = \frac{x_1 y_1 + \cdots + x_n y_n}{\sqrt{x_1^2 + \cdots + x_n^2}\sqrt{y_1^2 + \cdots + y_n^2}} \tag{2.1}$$

在式 (2.1) 中，分子是向量内积，分母是各个向量的范数。范数表示向量的大小，这里计算的是 L2 范数（即向量各个元素的平方和的平方根）。式 (2.1) 的要点是先对向量进行正规化，再求它们的内积。



余弦相似度直观地表示了“两个向量在多大程度上指向同一方向”。两个向量完全指向相同的方向时，余弦相似度为 1；完全指向相反的方向时，余弦相似度为 −1。

现在，我们来实现余弦相似度。基于式 (2.1)，代码如下所示。

```
def cos_similarity(x, y):
    nx = x / np.sqrt(np.sum(x**2)) # x的正规化
    ny = y / np.sqrt(np.sum(y**2)) # y的正规化
    return np.dot(nx, ny)
```

这里，我们假定参数 x 和 y 是 NumPy 数组。首先对向量进行正规化，然后求两个向量的内积。这里余弦相似度的实现虽然完成了，但是还有一个问题。那就是当零向量（元素全部为 0 的向量）被赋值给参数时，会出现“除数为 0”（zero division）的错误。

解决此类问题的一个常用方法是，在执行除法时加上一个微小值。这里，通过参数指定一个微小值 eps（eps 是 epsilon 的缩写），并默认 eps=1e-8（= 0.000 000 01）。这样修改后的余弦相似度的实现如下所示（✏ common/util.py）。

```
def cos_similarity(x, y, eps=1e-8):
    nx = x / (np.sqrt(np.sum(x ** 2)) + eps)
    ny = y / (np.sqrt(np.sum(y ** 2)) + eps)
    return np.dot(nx, ny)
```



这里我们用了 1e-8 作为微小值，在这么小的值的情况下，根据浮点数的舍入误差，这个微小值会被其他值“吸收”掉。在上面的实现中，因为这个微小值会被向量的范数“吸收”掉，所以在绝大多数情况下，加上 eps 不会对最终的计算结果造成影响。而当向量的范数为 0 时，这个微小值可以防止“除数为 0”的错误。

利用这个函数，可以如下求得单词向量间的相似度。这里，我们尝试求 you 和 i（= I）的相似度（✏ ch02/similarity.py）。

```
import sys
sys.path.append('..')
from common.util import preprocess, create_co_matrix, cos_similarity
```

```
text = 'You say goodbye and I say hello.'
corpus, word_to_id, id_to_word = preprocess(text)
vocab_size = len(word_to_id)
C = create_co_matrix(corpus, vocab_size)

c0 = C[word_to_id['you']]  # you的单词向量
c1 = C[word_to_id['i']]    # i的单词向量
print(cos_similarity(c0, c1))
# 0.7071067691154799
```

从上面的结果可知，you 和 i 的余弦相似度是 0.70 . . .。由于余弦相似度的取值范围是 −1 到 1，所以可以说这个值是相对比较高的（存在相似性）。

2.3.6　相似单词的排序

余弦相似度已经实现好了，使用这个函数，我们可以实现另一个便利的函数：当某个单词被作为查询词时，将与这个查询词相似的单词按降序显示出来。这里将这个函数称为 most_similar()，通过下列参数进行实现（表 2-1）。

```
most_similar(query, word_to_id, id_to_word, word_matrix, top=5)
```

表 2-1　most_similar() 函数的参数

参数名	说明
query	查询词
word_to_id	单词到单词 ID 的字典
id_to_word	单词 ID 到单词的字典
word_matrix	汇总了单词向量的矩阵，假定保存了与各行对应的单词向量
top	显示到前几位

这里我们直接给出 most_similar() 函数的实现，如下所示（✏ common/util.py）。

```
def most_similar(query, word_to_id, id_to_word, word_matrix, top=5):
    # ❶取出查询词
    if query not in word_to_id:
        print('%s is not found' % query)
        return
```

```
    print('\n[query] ' + query)
    query_id = word_to_id[query]
    query_vec = word_matrix[query_id]

    # ❷计算余弦相似度
    vocab_size = len(id_to_word)
    similarity = np.zeros(vocab_size)
    for i in range(vocab_size):
        similarity[i] = cos_similarity(word_matrix[i], query_vec)

    # ❸基于余弦相似度，按降序输出值
    count = 0
    for i in (-1 * similarity).argsort():
        if id_to_word[i] == query:
            continue
        print(' %s: %s' % (id_to_word[i], similarity[i]))

        count += 1
        if count >= top:
            return
```

上述实现按如下顺序执行。

❶ 取出查询词的单词向量。

❷ 分别求得查询词的单词向量和其他所有单词向量的余弦相似度。

❸ 基于余弦相似度的结果，按降序显示它们的值。

我们仅对步骤❸进行补充说明。在步骤❸中，将 `similarity` 数组中的元素索引按降序重新排列，并输出顶部的单词。这里使用 `argsort()` 方法对数组的索引进行了重排。这个 `argsort()` 方法可以按升序对 NumPy 数组的元素进行排序（不过，返回值是数组的索引）。下面是一个例子。

```
>>> x = np.array([100, -20, 2])
>>> x.argsort()
array([1, 2, 0])
```

上述代码对 NumPy 数组 `[100,-20,2]` 的各个元素按升序进行了排列。

此时，返回的数组的各个元素对应原数组的索引。上述结果的顺序是“第 1 个元素（−20）”“第 2 个元素（2）”“第 0 个元素（100）”。现在我们想做的是将单词的相似度按降序排列，因此，将 NumPy 数组的各个元素乘以 −1 后，再使用 `argsort()` 方法。接着上面的例子，有如下代码。

```
>>> (-x).argsort()
array([0, 2, 1])
```

使用这个 `argsort()`，可以按降序输出单词相似度。以上就是 `most_similar()` 函数的实现，下面我们来试着使用一下。这里将 you 作为查询词，显示与其相似的单词，代码如下所示（✏ ch02/most_similar.py）。

```
import sys
sys.path.append('..')
from common.util import preprocess, create_co_matrix, most_similar


text = 'You say goodbye and I say hello.'
corpus, word_to_id, id_to_word = preprocess(text)
vocab_size = len(word_to_id)
C = create_co_matrix(corpus, vocab_size)

most_similar('you', word_to_id, id_to_word, C, top=5)
```

执行代码后，会得到如下结果。

```
[query] you
 goodbye: 0.7071067691154799
 i: 0.7071067691154799
 hello: 0.7071067691154799
 say: 0.0
 and: 0.0
```

这个结果只按降序显示了 you 这个查询词的前 5 个相似单词，各个单词旁边的值是余弦相似度。观察上面的结果可知，和 you 最接近的单词有 3 个，分别是 goodbye、i（= I）和 hello。因为 i 和 you 都是人称代词，所以二者相似可以理解。但是，goodbye 和 hello 的余弦相似度也很高，这和我

们的感觉存在很大的差异。一个可能的原因是，这里的语料库太小了。后面我们会用更大的语料库进行相同的实验。

如上所述，我们通过共现矩阵成功地将单词表示为了向量。至此，基于计数的方法的基本内容就介绍完了。之所以说“基本”，是因为还有许多事情需要讨论。下一节，我们将说明当前方法的改进思路，并实现这个改进思路。

2.4 基于计数的方法的改进

上一节我们创建了单词的共现矩阵，并使用它成功地将单词表示为了向量。但是，这个共现矩阵还有许多可以改进的地方。本节我们将对其进行改进，并使用更实用的语料库，获得单词的“真实的”分布式表示。

2.4.1 点互信息

上一节的共现矩阵的元素表示两个单词同时出现的次数。但是，这种“原始”的次数并不具备好的性质。如果我们看一下高频词汇（出现次数很多的单词），就能明白其原因了。

比如，我们来考虑某个语料库中 the 和 car 共现的情况。在这种情况下，我们会看到很多“...the car...”这样的短语。因此，它们的共现次数将会很大。另外，car 和 drive 也明显有很强的相关性。但是，如果只看单词的出现次数，那么与 drive 相比，the 和 car 的相关性更强。这意味着，仅仅因为 the 是个常用词，它就被认为与 car 有很强的相关性。

为了解决这一问题，可以使用点互信息（Pointwise Mutual Information，PMI）这一指标。对于随机变量 x 和 y，它们的 PMI 定义如下（关于概率，将在 3.5.1 节详细说明）：

$$\mathrm{PMI}(x, y) = \log_2 \frac{P(x, y)}{P(x)P(y)} \tag{2.2}$$

其中，$P(x)$ 表示 x 发生的概率，$P(y)$ 表示 y 发生的概率，$P(x, y)$ 表示 x 和 y 同时发生的概率。PMI 的值越高，表明相关性越强。

在自然语言的例子中，$P(x)$ 就是指单词 x 在语料库中出现的概率。假设某个语料库中有 10 000 个单词，其中单词 the 出现了 100 次，则 $P(\texttt{"the"}) = \frac{100}{10\,000} = 0.01$。另外，$P(x, y)$ 表示单词 x 和 y 同时出现的概率。假设 the 和 car 一起出现了 10 次，则$P(\texttt{"the"}, \texttt{"car"}) = \frac{10}{10\,000} = 0.001$。

现在，我们使用共现矩阵（其元素表示单词共现的次数）来重写式 (2.2)。这里，将共现矩阵表示为 $\boldsymbol{C}$，将单词 x 和 y 的共现次数表示为 $\boldsymbol{C}(x, y)$，将单词 x 和 y 的出现次数分别表示为 $\boldsymbol{C}(x)$、$\boldsymbol{C}(y)$，将语料库的单词数量记为 N，则式 (2.2) 可以重写为：

$$\mathrm{PMI}(x, y) = \log_2 \frac{P(x, y)}{P(x)P(y)} = \log_2 \frac{\frac{\boldsymbol{C}(x,y)}{N}}{\frac{\boldsymbol{C}(x)}{N}\frac{\boldsymbol{C}(y)}{N}} = \log_2 \frac{\boldsymbol{C}(x, y) \cdot N}{\boldsymbol{C}(x)\boldsymbol{C}(y)} \tag{2.3}$$

根据式 (2.3)，可以由共现矩阵求 PMI。下面我们来具体地算一下。这里假设语料库的单词数量（N）为 10 000，the 出现 100 次，car 出现 20 次，drive 出现 10 次，the 和 car 共现 10 次，car 和 drive 共现 5 次。这时，如果从共现次数的角度来看，则与 drive 相比，the 和 car 的相关性更强。而如果从 PMI 的角度来看，结果是怎样的呢？我们来计算一下。

$$\mathrm{PMI}(\texttt{"the"}, \texttt{"car"}) = \log_2 \frac{10 \cdot 10\,000}{1000 \cdot 20} \approx 2.32 \tag{2.4}$$

$$\mathrm{PMI}(\texttt{"car"}, \texttt{"drive"}) = \log_2 \frac{5 \cdot 10\,000}{20 \cdot 10} \approx 7.97 \tag{2.5}$$

结果表明，在使用 PMI 的情况下，与 the 相比，drive 和 car 具有更强的相关性。这是我们想要的结果。之所以出现这个结果，是因为我们考虑了单词单独出现的次数。在这个例子中，因为 the 本身出现得多，所以 PMI 的得分被拉低了。式中的“≈”（near equal）表示近似相等的意思。

虽然我们已经获得了PMI这样一个好的指标，但是PMI也有一个问题。

那就是当两个单词的共现次数为 0 时，$\log_2 0 = -\infty$。为了解决这个问题，实践上我们会使用下述正的点互信息（Positive PMI，PPMI）。

$$\text{PPMI}(x, y) = \max(0, \text{PMI}(x, y)) \tag{2.6}$$

根据式 (2.6)，当 PMI 是负数时，将其视为 0，这样就可以将单词间的相关性表示为大于等于 0 的实数。下面，我们来实现将共现矩阵转化为 PPMI 矩阵的函数。我们把这个函数称为 `ppmi(C, verbose=False, eps=1e-8)`（✏ `common/util.py`）。

```python
def ppmi(C, verbose=False, eps=1e-8):
    M = np.zeros_like(C, dtype=np.float32)
    N = np.sum(C)
    S = np.sum(C, axis=0)
    total = C.shape[0] * C.shape[1]
    cnt = 0

    for i in range(C.shape[0]):
        for j in range(C.shape[1]):
            pmi = np.log2(C[i, j] * N / (S[j]*S[i]) + eps)
            M[i, j] = max(0, pmi)

            if verbose:
                cnt += 1
                if cnt % (total//100+1) == 0:
                    print('%.1f%% done' % (100*cnt/total))
    return M
```

这里，参数 `C` 表示共现矩阵，`verbose` 是决定是否输出运行情况的标志。当处理大语料库时，设置 `verbose=True`，可以用于确认运行情况。在这段代码中，为了仅从共现矩阵求 PPMI 矩阵而进行了简单的实现。具体来说，当单词 x 和 y 的共现次数为 $\boldsymbol{C}(x, y)$ 时，$\boldsymbol{C}(x) = \sum_i \boldsymbol{C}(i, x)$，$\boldsymbol{C}(y) = \sum_i \boldsymbol{C}(i, y)$，$N = \sum_i \sum_j \boldsymbol{C}(i, j)$，进行这样近似并实现。另外，在上述代码中，为了防止 `np.log2(0)=-inf` 而使用了微小值 `eps`。



在2.3.5节中，为了防止“除数为0”的错误，我们给分母添加了一个微小值。这里也一样，通过将np.log(x)改为np.log(x + eps)，可以防止对数运算发散到负无穷大。

现在将共现矩阵转化为PPMI矩阵，可以像下面这样进行实现（✏ ch02/ppmi.py）。

```
import sys
sys.path.append('..')
import numpy as np
from common.util import preprocess, create_co_matrix, cos_similarity,
 ↪   ppmi


text = 'You say goodbye and I say hello.'
corpus, word_to_id, id_to_word = preprocess(text)
vocab_size = len(word_to_id)
C = create_co_matrix(corpus, vocab_size)
W = ppmi(C)

np.set_printoptions(precision=3) # 有效位数为3位
print('covariance matrix')
print(C)
print('-'*50)
print('PPMI')
print(W)
```

运行该文件，可以得到下述结果。

```
covariance matrix
[[0 1 0 0 0 0 0]
 [1 0 1 0 1 1 0]
 [0 1 0 1 0 0 0]
 [0 0 1 0 1 0 0]
 [0 1 0 1 0 0 0]
 [0 1 0 0 0 0 1]
 [0 0 0 0 0 1 0]]
--------------------------------------------------
```

```
PPMI
[[ 0.     1.807  0.     0.     0.     0.     0.    ]
 [ 1.807  0.     0.807  0.     0.807  0.807  0.    ]
 [ 0.     0.807  0.     1.807  0.     0.     0.    ]
 [ 0.     0.     1.807  0.     1.807  0.     0.    ]
 [ 0.     0.807  0.     1.807  0.     0.     0.    ]
 [ 0.     0.807  0.     0.     0.     0.     2.807]
 [ 0.     0.     0.     0.     0.     2.807  0.    ]]
```

这样一来，我们就将共现矩阵转化为了 PPMI 矩阵。此时，PPMI 矩阵的各个元素均为大于等于 0 的实数。我们得到了一个由更好的指标形成的矩阵，这相当于获取了一个更好的单词向量。

但是，这个 PPMI 矩阵还是存在一个很大的问题，那就是随着语料库的词汇量增加，各个单词向量的维数也会增加。如果语料库的词汇量达到 10 万，则单词向量的维数也同样会达到 10 万。实际上，处理 10 万维向量是不现实的。

另外，如果我们看一下这个矩阵，就会发现其中很多元素都是 0。这表明向量中的绝大多数元素并不重要，也就是说，每个元素拥有的“重要性”很低。另外，这样的向量也容易受到噪声影响，稳健性差。对于这些问题，一个常见的方法是向量降维。

2.4.2　降维

所谓降维（dimensionality reduction），顾名思义，就是减少向量维度。但是，并不是简单地减少，而是在尽量保留“重要信息”的基础上减少。如图 2-8 所示，我们要观察数据的分布，并发现重要的“轴”。

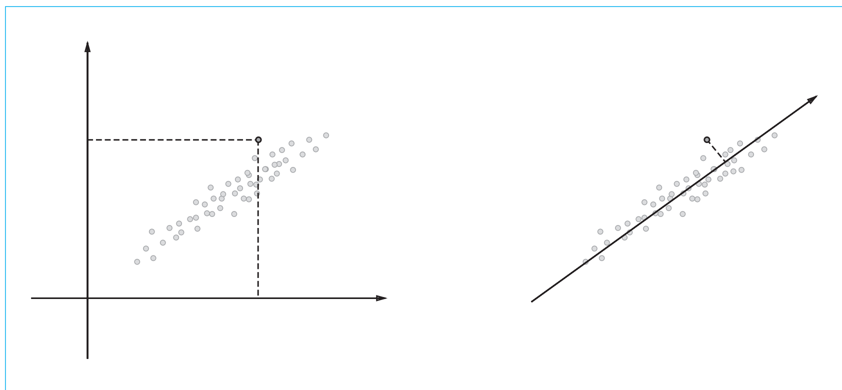


图2-8　降维示意图：发现重要的轴（数据分布广的轴），将二维数据表示为一维数据

在图 2-8 中，考虑到数据的广度，导入了一根新轴，以将原来用二维坐标表示的点表示在一个坐标轴上。此时，用新轴上的投影值来表示各个数据点的值。这里非常重要的一点是，选择新轴时要考虑数据的广度。如此，仅使用一维的值也能捕获数据的本质差异。在多维数据中，也可以进行同样的处理。



向量中的大多数元素为 0 的矩阵（或向量）称为稀疏矩阵（或稀疏向量）。这里的重点是，从稀疏向量中找出重要的轴，用更少的维度对其进行重新表示。结果，稀疏矩阵就会被转化为大多数元素均不为 0 的密集矩阵。这个密集矩阵就是我们想要的单词的分布式表示。

降维的方法有很多，这里我们使用奇异值分解（Singular Value Decomposition，SVD）。SVD 将任意矩阵分解为 3 个矩阵的乘积，如下式所示：

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^{\mathrm{T}} \tag{2.7}$$

如式 (2.7) 所示，SVD 将任意的矩阵 $\boldsymbol{X}$ 分解为 $\boldsymbol{U}$、$\boldsymbol{S}$、$\boldsymbol{V}$ 这 3 个矩阵的乘积，其中 $\boldsymbol{U}$ 和 $\boldsymbol{V}$ 是列向量彼此正交的正交矩阵，$\boldsymbol{S}$ 是除了对角线元素以外其余元素均为 0 的对角矩阵。图 2-9 中直观地表示出了这些矩阵。

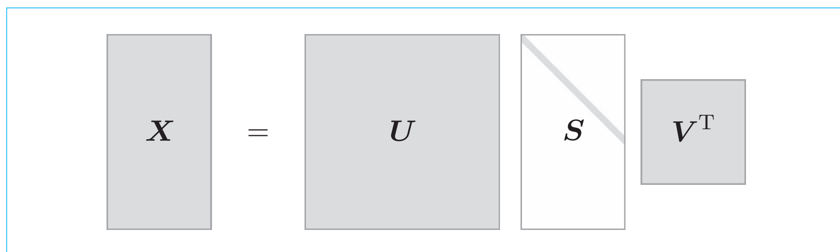


图 2-9 基于SVD的矩阵变换（白色部分表示元素为0）

在式 (2.7) 中，$\boldsymbol{U}$ 是正交矩阵。这个正交矩阵构成了一些空间中的基轴（基向量），我们可以将矩阵 $\boldsymbol{U}$ 作为“单词空间”。$\boldsymbol{S}$ 是对角矩阵，奇异值在对角线上降序排列。简单地说，我们可以将奇异值视为“对应的基轴”的重要性。这样一来，如图 2-10 所示，减少非重要元素就成为可能。

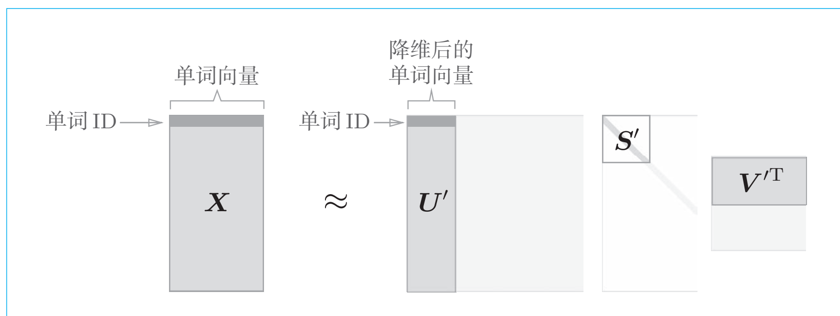


图 2-10 基于SVD的降维示意图

如图 2-10 所示，矩阵 $\boldsymbol{S}$ 的奇异值小，对应的基轴的重要性低，因此，可以通过去除矩阵 $\boldsymbol{U}$ 中的多余的列向量来近似原始矩阵。用我们正在处理的“单词的 PPMI 矩阵”来说明的话，矩阵 $\boldsymbol{X}$ 的各行包含对应的单词 ID 的单词向量，这些单词向量使用降维后的矩阵 $\boldsymbol{U}'$ 表示。



单词的共现矩阵是正方形矩阵，但在图2-10中，为了和之前的图一致，画的是长方形。另外，这里对SVD的介绍仅限于最直观的概要性的说明。想从数学角度仔细理解的读者，请参考文献 [20] 等。

2.4.3 基于SVD的降维

接下来，我们使用 Python 来实现 SVD，这里可以使用 NumPy 的 linalg 模块中的 svd 方法。linalg 是 linear algebra（线性代数）的简称。下面，我们创建一个共现矩阵，将其转化为 PPMI 矩阵，然后对其进行 SVD（✏ ch02/count_method_small.py）。

```
import sys
sys.path.append('..')
import numpy as np
import matplotlib.pyplot as plt
from common.util import preprocess, create_co_matrix, ppmi


text = 'You say goodbye and I say hello.'
corpus, word_to_id, id_to_word = preprocess(text)
vocab_size = len(id_to_word)
C = create_co_matrix(corpus, vocab_size, window_size=1)
W = ppmi(C)

# SVD
U, S, V = np.linalg.svd(W)
```

SVD 执行完毕。上面的变量 U 包含经过 SVD 转化的密集向量表示。现在，我们来看一下它的内容。单词 ID 为 0 的单词向量如下。

```
print(C[0])  # 共现矩阵
# [0 1 0 0 0 0 0]

print(W[0])  # PPMI矩阵
# [ 0.     1.807  0.     0.     0.     0.     0.   ]

print(U[0])  # SVD
# [ 3.409e-01 -1.110e-16 -1.205e-01 -4.441e-16  0.000e+00 -9.323e-01
#   2.226e-16]
```

如上所示，原先的稀疏向量 W[0] 经过 SVD 被转化成了密集向量 U[0]。

如果要对这个密集向量降维，比如把它降维到二维向量，取出前两个元素即可。

```
print(U[0, :2])
# [ 3.409e-01 -1.110e-16]
```

这样我们就完成了降维。现在，我们用二维向量表示各个单词，并把它们画在图上，代码如下。

```
for word, word_id in word_to_id.items():
    plt.annotate(word, (U[word_id, 0], U[word_id, 1]))

plt.scatter(U[:,0], U[:,1], alpha=0.5)
plt.show()
```

`plt.annotate(word, x, y)` 函数在 2D 图形中坐标为 `(x,y)` 的地方绘制单词的文本。执行上述代码，结果如图 2-11 所示[①]。

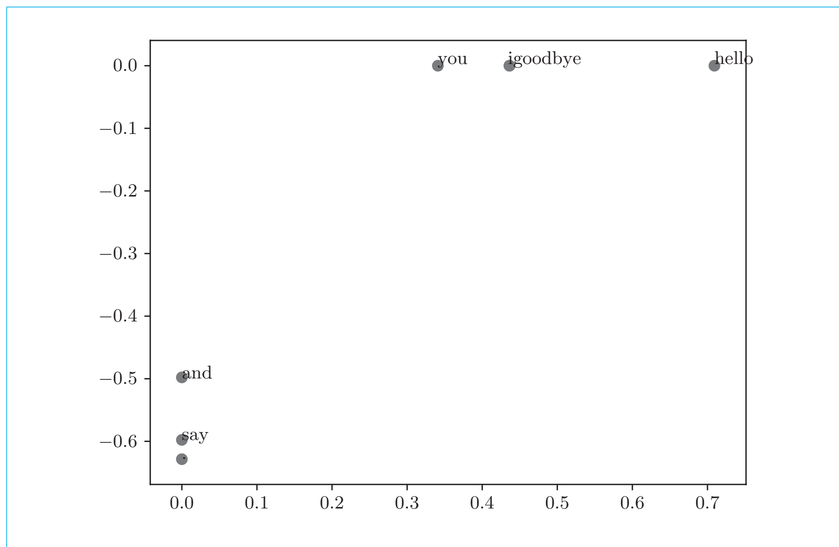


图 2-11 对共现矩阵执行 SVD，并在图上绘制各个单词的二维向量（i 和 goodbye 重叠）

① 根据操作系统的种类或 Matplotlib 版本的不同，输出的图可能和图 2-11 所有不同。

观察该图可以发现，goodbye 和 hello、you 和 i 位置接近，这是比较符合我们的直觉的。但是，因为我们使用的语料库很小，有些结果就比较微妙。下面，我们将使用更大的 PTB 数据集进行相同的实验。首先，我们简单介绍一下 PTB 数据集。



如果矩阵大小是 N，SVD 的计算的复杂度将达到 $O(N^3)$。这意味着 SVD 需要与 N 的立方成比例的计算量。因为现实中这样的计算量是做不到的，所以往往会使用 Truncated SVD[21] 等更快的方法。Truncated SVD 通过截去（truncated）奇异值较小的部分，从而实现高速化。下一节，作为另一个选择，我们将使用 sklearn 库的 Truncated SVD。

2.4.4 PTB 数据集

到目前为止，我们使用了非常小的文本数据作为语料库。这里，我们将使用一个大小合适的“真正的”语料库——Penn Treebank 语料库（以下简称为 PTB）。



PTB 语料库经常被用作评价提案方法的基准。本书中我们将使用 PTB 语料库进行各种实验。

我们使用的 PTB 语料库在 word2vec 的发明者托马斯·米科洛夫（Tomas Mikolov）的网页上有提供。这个 PTB 语料库是以文本文件的形式提供的，与原始的 PTB 的文章相比，多了若干预处理，包括将稀有单词替换成特殊字符 <unk>（unk 是 unknown 的简称），将具体的数字替换成“N”等。下面，我们将经过这些预处理之后的文本数据作为 PTB 语料库使用。作为参考，图 2-12 给出了 PTB 语料库的部分内容。

如图 2-12 所示，在 PTB 语料库中，一行保存一个句子。在本书中，我们将所有句子连接起来，并将其视为一个大的时序数据。此时，在每个句子的结尾处插入一个特殊字符 <eos>（eos 是 end of sentence 的简称）。

```
1 consumers may want to move their telephones a little closer to the tv set
2 <unk> <unk> watching abc 's monday night football can now vote during <unk> for the greatest play in N years from
- among four or five <unk> <unk>
3 two weeks ago viewers of several nbc <unk> consumer segments started calling a N number for advice on various
- <unk> issues
4 and the new syndicated reality show hard copy records viewers ' opinions for possible airing on the next day 's show
5 interactive telephone technology has taken a new leap in <unk> and television programmers are racing to exploit the
- possibilities
6 eventually viewers may grow <unk> with the technology and <unk> the cost
```

图2-12　PTB语料库（文本文件）的例子



本书不考虑句子的分割，将多个句子连接起来得到的内容视为一个大的时序数据。当然，也可以以句子为单位进行处理，比如，以句子为单位计算词频。不过，考虑到简单性，本书不进行以句子为单位的处理。

在本书中，为了方便使用 Penn Treebank 数据集，我们准备了专门的 Python 代码。这个文件在 `dataset/ptb.py` 中，并假定从章节目录（`ch01`、`ch02`、…）使用。比如，我们将当前目录移到 `ch02` 目录，并在这个目录中调用 `python show_ptb.py`。使用 `ptb.py` 的例子如下所示（✏ `ch02/show_ptb.py`）。

```python
import sys
sys.path.append('..')
from dataset import ptb


corpus, word_to_id, id_to_word = ptb.load_data('train')

print('corpus size:', len(corpus))
print('corpus[:30]:', corpus[:30])
print()
print('id_to_word[0]:', id_to_word[0])
print('id_to_word[1]:', id_to_word[1])
print('id_to_word[2]:', id_to_word[2])
print()
print("word_to_id['car']:", word_to_id['car'])
```

```
print("word_to_id['happy']:", word_to_id['happy'])
print("word_to_id['lexus']:", word_to_id['lexus'])
```

后面再具体解释这段代码，我们先来看一下它的执行结果。

```
corpus size: 929589
corpus[:30]: [ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
19 20 21 22 23
 24 25 26 27 28 29]

id_to_word[0]: aer
id_to_word[1]: banknote
id_to_word[2]: berlitz

word_to_id['car']: 3856
word_to_id['happy']: 4428
word_to_id['lexus']: 7426
```

语料库的用法和之前一样。corpus 中保存了单词 ID 列表，id_to_word 是将单词 ID 转化为单词的字典，word_to_id 是将单词转化为单词 ID 的字典。

如上面的代码所示，使用 ptb.load_data() 加载数据。此时，指定参数 'train'、'test' 和 'valid' 中的一个，它们分别对应训练用数据、测试用数据和验证用数据中的一个。以上就是 ptb.py 文件的使用方法。

2.4.5　基于PTB数据集的评价

下面，我们将基于计数的方法应用于 PTB 数据集。这里建议使用更快速的 SVD 对大矩阵执行 SVD，为此我们需要安装 sklearn 模块。当然，虽然仍可以使用基本版的 SVD（np.linalg.svd()），但是这需要更多的时间和内存。我们把源代码一并给出，如下所示（✏ ch02/count_method_big.py）。

```
import sys
sys.path.append('..')
import numpy as np
from common.util import most_similar, create_co_matrix, ppmi
from dataset import ptb
```

```
window_size = 2
wordvec_size = 100

corpus, word_to_id, id_to_word = ptb.load_data('train')
vocab_size = len(word_to_id)
print('counting  co-occurrence ...')
C = create_co_matrix(corpus, vocab_size, window_size)
print('calculating PPMI ...')
W = ppmi(C, verbose=True)

print('calculating SVD ...')
try:
    # truncated SVD (fast!)
    from sklearn.utils.extmath import randomized_svd
    U, S, V = randomized_svd(W, n_components=wordvec_size, n_iter=5,
                             random_state=None)
except ImportError:
    # SVD (slow)
    U, S, V = np.linalg.svd(W)

word_vecs = U[:, :wordvec_size]

querys = ['you', 'year', 'car', 'toyota']
for query in querys:
    most_similar(query, word_to_id, id_to_word, word_vecs, top=5)
```

这里，为了执行 SVD，我们使用了 sklearn 的 `randomized_svd()` 方法。该方法通过使用了随机数的 Truncated SVD，仅对奇异值较大的部分进行计算，计算速度比常规的 SVD 快。剩余的代码和之前使用小语料库时的代码差不太多。执行代码，可以得以下结果（因为使用了随机数，所以在使用 Truncated SVD 的情况下，每次的结果都不一样）。

```
[query] you
 i: 0.702039909619
 we: 0.699448543998
 've: 0.554828709147
 do: 0.534370693098
 else: 0.512044146526
```

```
[query] year
 month: 0.731561990308
 quarter: 0.658233992457
 last: 0.622425716735
 earlier: 0.607752074689
 next: 0.601592506413

[query] car
 luxury: 0.620933665528
 auto: 0.615559874277
 cars: 0.569818364381
 vehicle: 0.498166879744
 corsica: 0.472616831915

[query] toyota
 motor: 0.738666107068
 nissan: 0.677577542584
 motors: 0.647163210589
 honda: 0.628862370943
 lexus: 0.604740429865
```

观察结果可知，首先，对于查询词 you，可以看到 i、we 等人称代词排在前面，这些都是在语法上具有相同用法的词。再者，查询词 year 有 month、quarter 等近义词，查询词 car 有 auto、vehicle 等近义词。此外，将 toyota 作为查询词时，出现了 nissan、honda 和 lexus 等汽车制造商名或者品牌名。像这样，在含义或语法上相似的单词表示为相近的向量，这符合我们的直觉。

我们终于成功地将单词含义编码成了向量，真是可喜可贺！使用语料库，计算上下文中的单词数量，将它们转化 PPMI 矩阵，再基于 SVD 降维获得好的单词向量。这就是单词的分布式表示，每个单词表示为固定长度的密集向量。

在本章的实验中，我们只看了一部分单词的近义词，但是可以确认许多其他的单词也有这样的性质。期待使用更大的语料库可以获得更好的单词的分布式表示！

2.5 小结

本章，我们以自然语言为对象，特别是以让计算机理解单词含义为主题展开了讨论。为了达到这一目标，我们介绍了基于同义词词典的方法，也考察了基于计数的方法。

使用基于同义词词典的方法，需要人工逐个定义单词之间的相关性。这样的工作非常费力，在表现力上也存在限制（比如，不能表示细微的差别）。而基于计数的方法从语料库中自动提取单词含义，并将其表示为向量。具体来说，首先创建单词的共现矩阵，将其转化为 PPMI 矩阵，再基于 SVD 降维以提高稳健性，最后获得每个单词的分布式表示。另外，我们已经确认过，这样的分布式表示具有在含义或语法上相似的单词在向量空间上位置相近的性质。

为了方便处理语料库的文本数据，我们实现了几个预处理函数。具体来说，包括测量向量间相似度的函数（`cos_similarity()`）、用于显示相似单词的排名的函数（`most_similar()`）。这些函数在后面的章节中还会用到。

本章所学的内容

- 使用 WordNet 等同义词词典，可以获取近义词或测量单词间的相似度等
- 使用同义词词典的方法存在创建词库需要大量人力、新词难更新等问题
- 目前，使用语料库对单词进行向量化是主流方法
- 近年来的单词向量化方法大多基于“单词含义由其周围的单词构成”这一分布式假设
- 在基于计数的方法中，对语料库中的每个单词周围的单词的出现频数进行计数并汇总（= 共现矩阵）
- 通过将共现矩阵转化为 PPMI 矩阵并降维，可以将大的稀疏向量转变为小的密集向量
- 在单词的向量空间中，含义上接近的单词距离上理应也更近

第3章 word2vec

"没有判断依据，就不要去推理。"

——阿瑟・柯南・道尔《波希米亚丑闻》(收录于《冒险史》)

接着上一章，本章的主题仍是单词的分布式表示。在上一章中，我们使用基于计数的方法得到了单词的分布式表示。本章我们将讨论该方法的替代方法，即基于推理的方法。

顾名思义，基于推理的方法使用了推理机制。当然，这里的推理机制用的是神经网络。本章，著名的 word2vec 将会登场。我们将花很多时间考察 word2vec 的结构，并通过代码实现来加深对它的理解。

本章的目标是实现一个简单的 word2vec。这个简单的 word2vec 会优先考虑易理解性，从而牺牲一定的处理效率。因此，我们不会用它来处理大规模数据集，但用它处理小数据集毫无问题。下一章我们会对这个简单的 word2vec 进行改进，从而完成一个"真正的"word2vec。现在，让我们一起进入基于推理的方法和 word2vec 的世界吧！

3.1 基于推理的方法和神经网络

用向量表示单词的研究最近正在如火如荼地展开，其中比较成功的方法大致可以分为两种：一种是基于计数的方法；另一种是基于推理的方法。

虽然两者在获得单词含义的方法上差别很大，但是两者的背景都是分布式假设。

本节我们将指出基于计数的方法的问题，并从宏观角度说明它的替代方法——基于推理的方法的优点。另外，为了做好 word2vec 的准备工作，我们会看一个用神经网络处理单词的例子。

3.1.1 基于计数的方法的问题

如上一章所说，基于计数的方法根据一个单词周围的单词的出现频数来表示该单词。具体来说，先生成所有单词的共现矩阵，再对这个矩阵进行 SVD，以获得密集向量（单词的分布式表示）。但是，基于计数的方法在处理大规模语料库时会出现问题。

在现实世界中，语料库处理的单词数量非常大。比如，据说英文的词汇量超过 100 万个。如果词汇量超过 100 万个，那么使用基于计数的方法就需要生成一个 100 万 × 100 万的庞大矩阵，但对如此庞大的矩阵执行 SVD 显然是不现实的。



对于一个 $n \times n$ 的矩阵，SVD 的复杂度是 $O(n^3)$，这表示计算量与 n 的立方成比例增长。如此大的计算成本，即便是超级计算机也无法胜任。实际上，利用近似方法和稀疏矩阵的性质，可以在一定程度上提高处理速度，但还是需要大量的计算资源和时间。

基于计数的方法使用整个语料库的统计数据（共现矩阵和 PPMI 等），通过一次处理（SVD 等）获得单词的分布式表示。而基于推理的方法使用神经网络，通常在 mini-batch 数据上进行学习。这意味着神经网络一次只需要看一部分学习数据（mini-batch），并反复更新权重。这种学习机制上的差异如图 3-1 所示。

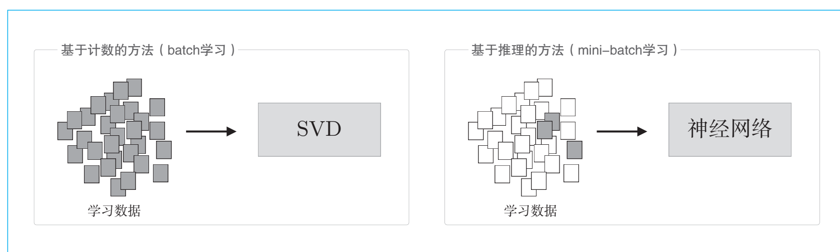


图 3-1 基于计数的方法和基于推理的方法的比较

如图 3-1 所示，基于计数的方法一次性处理全部学习数据；反之，基于推理的方法使用部分学习数据逐步学习。这意味着，在词汇量很大的语料库中，即使 SVD 等的计算量太大导致计算机难以处理，神经网络也可以在部分数据上学习。并且，神经网络的学习可以使用多台机器、多个 GPU 并行执行，从而加速整个学习过程。在这方面，基于推理的方法更有优势。

基于推理的方法和基于计数的方法相比，还有一些其他的优点。关于这一点，在详细说明基于推理的方法（特别是 word2vec）之后，我们会在 3.5.3 节再次讨论。

3.1.2 基于推理的方法的概要

基于推理的方法的主要操作是“推理”。如图 3-2 所示，当给出周围的单词（上下文）时，预测“?”处会出现什么单词，这就是推理。



图 3-2 基于两边的单词（上下文），预测“？”处出现什么单词

解开图 3-2 中的推理问题并学习规律，就是基于推理的方法的主要任务。通过反复求解这些推理问题，可以学习到单词的出现模式。从“模型视角”出发，这个推理问题如图 3-3 所示。

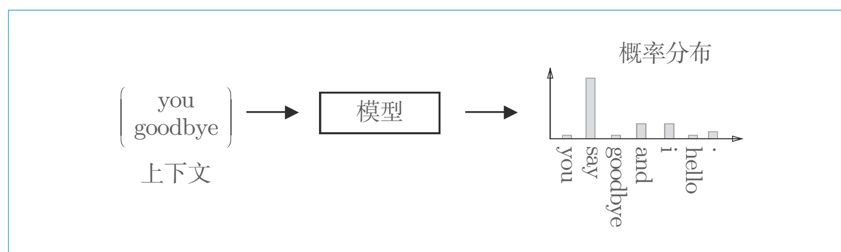


图3-3 基于推理的方法：输入上下文，模型输出各个单词的出现概率

如图 3-3 所示，基于推理的方法引入了某种模型，我们将神经网络用于此模型。这个模型接收上下文信息作为输入，并输出（可能出现的）各个单词的出现概率。在这样的框架中，使用语料库来学习模型，使之能做出正确的预测。另外，作为模型学习的产物，我们得到了单词的分布式表示。这就是基于推理的方法的全貌。



基于推理的方法和基于计数的方法一样，也基于分布式假设。分布式假设假设“单词含义由其周围的单词构成”。基于推理的方法将这一假设归结为了上面的预测问题。由此可见，不管是哪种方法，如何对基于分布式假设的“单词共现”建模都是最重要的研究主题。

3.1.3 神经网络中单词的处理方法

从现在开始，我们将使用神经网络来处理单词。但是，神经网络无法直接处理 you 或 say 这样的单词，要用神经网络处理单词，需要先将单词转化为固定长度的向量。对此，一种方式是将单词转换为 one-hot 表示（one-hot 向量）。在 one-hot 表示中，只有一个元素是 1，其他元素都是 0。

我们来看一个 one-hot 表示的例子。和上一章一样，我们用“You say goodbye and I say hello.”这个一句话的语料库来说明。在这个语料库中，一共有 7 个单词（“you”“say”“goodbye”“and”“i”“hello”“.”）。此时，各个单词可以转化为图 3-4 所示的 one-hot 表示。

单词	单词ID	one–hot表示
you goodbye	0 2	(1, 0, 0, 0, 0, 0, 0) (0, 0, 1, 0, 0, 0, 0)

图 3-4 单词、单词 ID 以及它们的 one-hot 表示

如图 3-4 所示，单词可以表示为文本、单词 ID 和 one-hot 表示。此时，要将单词转化为 one-hot 表示，就需要准备元素个数与词汇个数相等的向量，并将单词 ID 对应的元素设为 1，其他元素设为 0。像这样，只要将单词转化为固定长度的向量，神经网络的输入层的神经元个数就可以固定下来（图 3-5）。

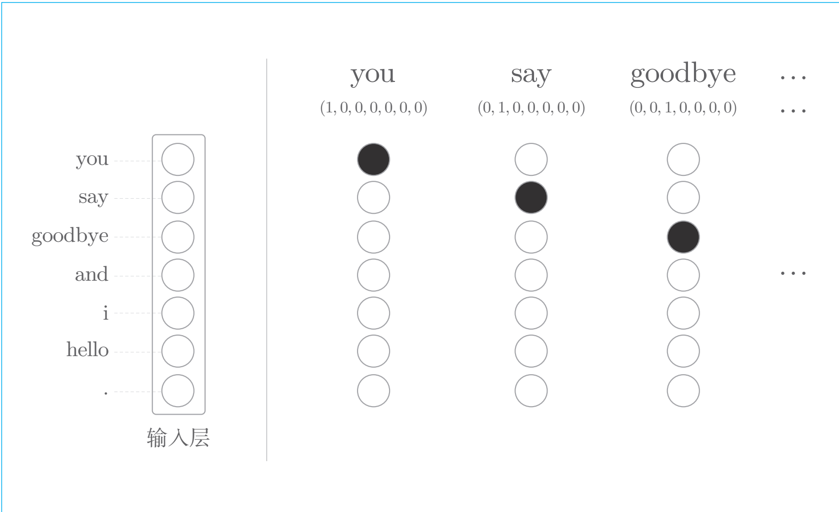


图 3-5 输入层的神经元：各个神经元对应于各个单词。图中神经元为 1 的地方用黑色绘制，为 0 的地方用白色绘制

如图 3-5 所示，输入层由 7 个神经元表示，分别对应于 7 个单词（第 1 个神经元对应于 you，第 2 个神经元对应于 say）。

现在事情变得很简单了。因为只要将单词表示为向量，这些向量就可以

由构成神经网络的各种“层”来处理。比如，对于one-hot表示的某个单词，使用全连接层对其进行变换的情况如图3-6所示。

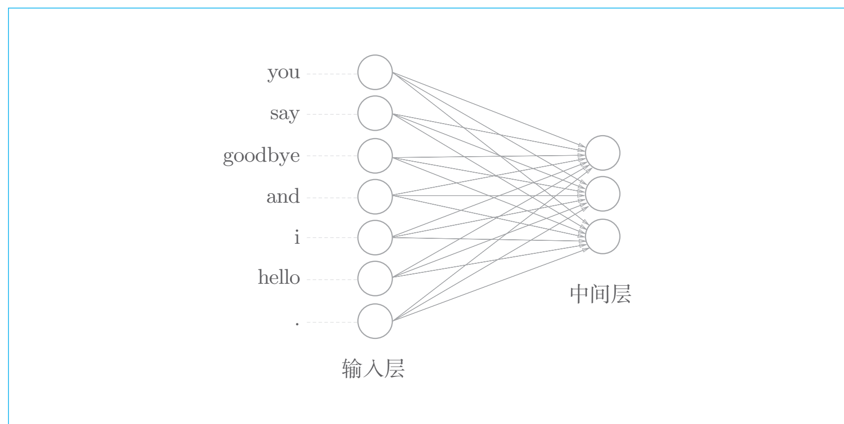


图3-6 基于神经网络的全连接层的变换：输入层的各个神经元分别对应于7个单词（中间层的神经元暂为3个）

如图3-6所示，全连接层通过箭头连接所有节点。这些箭头拥有权重（参数），它们和输入层神经元的加权和成为中间层的神经元。另外，本章使用的全连接层将省略偏置（这是为了配合后文对word2vec的说明）。



没有偏置的全连接层相当于在计算矩阵乘积。在很多深度学习的框架中，在生成全连接层时，都可以选择不使用偏置。在本书中，不使用偏置的全连接层相当于MatMul层（该层已经在第1章中实现）。

在图3-6中，神经元之间的连接是用箭头表示的。之后，为了明确地显示权重，我们将使用图3-7所示的方法。

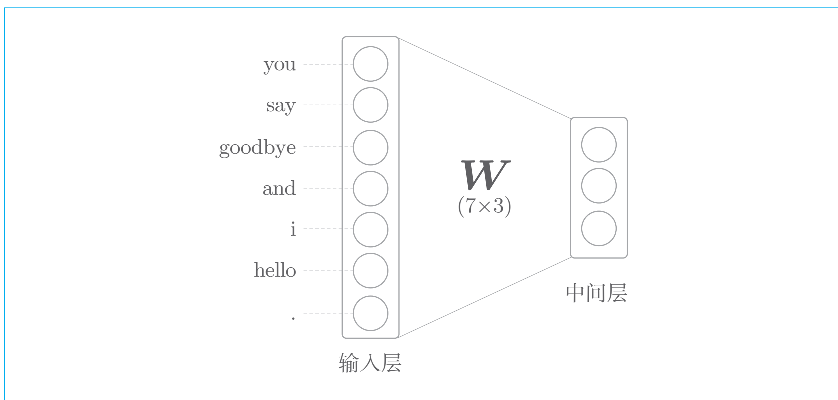


图 3-7 基于全连接层的变换的简化图示：将全连接层的权重表示为一个 7×3 形状的 $\boldsymbol{W}$ 矩阵

现在，我们看一下代码。这里的全连接层变换可以写成如下的 Python 代码。

```
import numpy as np

c = np.array([[1, 0, 0, 0, 0, 0, 0]])  # 输入
W = np.random.randn(7, 3)              # 权重
h = np.dot(c, W)                       # 中间节点
print(h)
# [[-0.70012195  0.25204755 -0.79774592]]
```

这段代码将单词 ID 为 0 的单词表示为了 one-hot 表示，并用全连接层对其进行了变换。作为复习，全连接层的计算通过矩阵乘积进行。这可以用 NumPy 的 `np.dot()` 来实现（省略偏置）。



这里，输入数据（变量 `c`）的维数（`ndim`）是 2。这是考虑了 mini-batch 处理，将各个数据保存在了第 1 维（0 维度）中。

希望读者注意一下 `c` 和 `W` 进行矩阵乘积计算的地方。此处，`c` 是 one-hot 表示，单词 ID 对应的元素是 1，其他地方都是 0。因此，如图 3-8 所示，

上述代码中的 `c` 和 `W` 的矩阵乘积相当于“提取”权重的对应行向量。

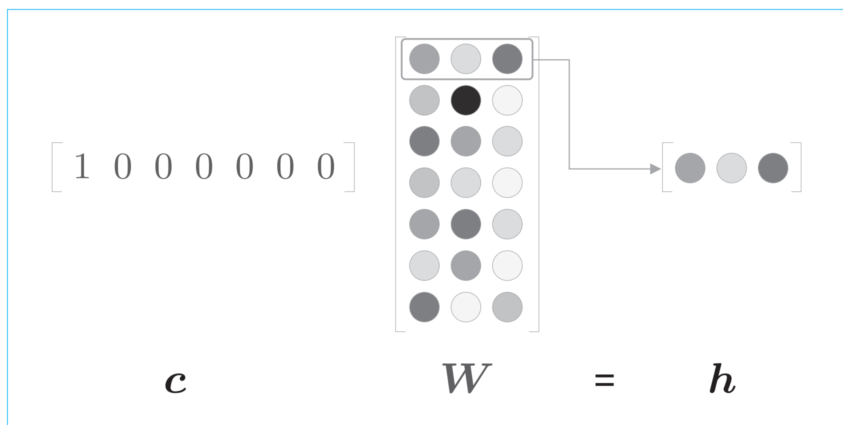


图3-8 在上下文 c 和权重 W 的矩阵乘积中，对应位置的行向量被提取（权重的各个元素的大小用灰度表示）

这里，仅为了提取权重的行向量而进行矩阵乘积计算好像不是很有效率。关于这一点，我们将在 4.1 节进行改进。另外，上述代码的功能也可以使用第 1 章中实现的 MatMul 层完成，如下所示。

```
import sys
sys.path.append('..')
import numpy as np
from common.layers import MatMul

c = np.array([[1, 0, 0, 0, 0, 0, 0]])
W = np.random.randn(7, 3)
layer = MatMul(W)
h = layer.forward(c)
print(h)
# [[-0.70012195  0.25204755 -0.79774592]]
```

这里，我们先导入了 `common` 目录下的 MatMul 层。之后，将 MatMul 层的权重设为了 `W`，并使用 `forward()` 方法执行正向传播。

3.2 简单的word2vec

上一节我们学习了基于推理的方法，并基于代码讨论了神经网络中单词的处理方法，至此准备工作就完成了，现在是时候实现 word2vec 了。

我们要做的事情是将神经网络嵌入到图 3-3 所示的模型中。这里，我们使用由原版 word2vec 提出的名为 continuous bag-of-words（CBOW）的模型作为神经网络。



word2vec一词最初用来指程序或者工具，但是随着该词的流行，在某些语境下，也指神经网络的模型。正确地说，CBOW模型和skip-gram模型是word2vec中使用的两个神经网络。本节我们将主要讨论CBOW模型。关于这两个模型的差异，我们将在3.5.2节详细介绍。

3.2.1 CBOW模型的推理

CBOW 模型是根据上下文预测目标词的神经网络（“目标词”是指中间的单词，它周围的单词是“上下文”）。通过训练这个 CBOW 模型，使其能尽可能地进行正确的预测，我们可以获得单词的分布式表示。

CBOW 模型的输入是上下文。这个上下文用 `['you', 'goodbye']` 这样的单词列表表示。我们将其转换为 one-hot 表示，以便 CBOW 模型可以进行处理。在此基础上，CBOW 模型的网络可以画成图 3-9 这样。

图 3-9 是 CBOW 模型的网络。它有两个输入层，经过中间层到达输出层。这里，从输入层到中间层的变换由相同的全连接层（权重为 $\boldsymbol{W}_{\text{in}}$）完成，从中间层到输出层神经元的变换由另一个全连接层（权重为 $\boldsymbol{W}_{\text{out}}$）完成。



这里，因为我们对上下文仅考虑两个单词，所以输入层有两个。如果对上下文考虑 N 个单词，则输入层会有 N 个。

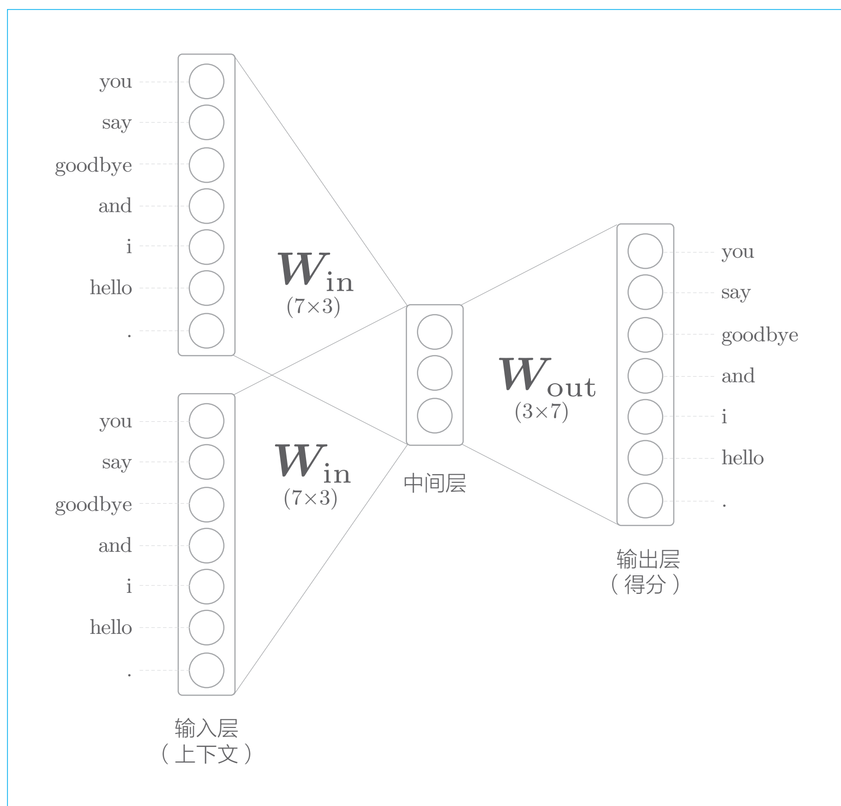


图 3-9 CBOW 模型的网络结构

现在，我们注意一下图 3-9 的中间层。此时，中间层的神经元是各个输入层经全连接层变换后得到的值的“平均”。就上面的例子而言，经全连接层变换后，第 1 个输入层转化为 $\boldsymbol{h}_1$，第 2 个输入层转化为 $\boldsymbol{h}_2$，那么中间层的神经元是 $\frac{1}{2}(\boldsymbol{h}_1+\boldsymbol{h}_2)$。

最后是图 3-9 中的输出层，这个输出层有 7 个神经元。这里重要的是，这些神经元对应于各个单词。输出层的神经元是各个单词的得分，它的值越大，说明对应单词的出现概率就越高。得分是指在被解释为概率之前的值，对这些得分应用 Softmax 函数，就可以得到概率。



有时将得分经过 Softmax 层之后的神经元称为输出层。这里，我们将输出得分的节点称为输出层。

如图 3-9 所示，从输入层到中间层的变换由全连接层（权重是 $\boldsymbol{W}_{\text{in}}$）完成。此时，全连接层的权重 $\boldsymbol{W}_{\text{in}}$ 是一个 7×3 的矩阵。提前剧透一下，这个权重就是我们要的单词的分布式表示，如图 3-10 所示。

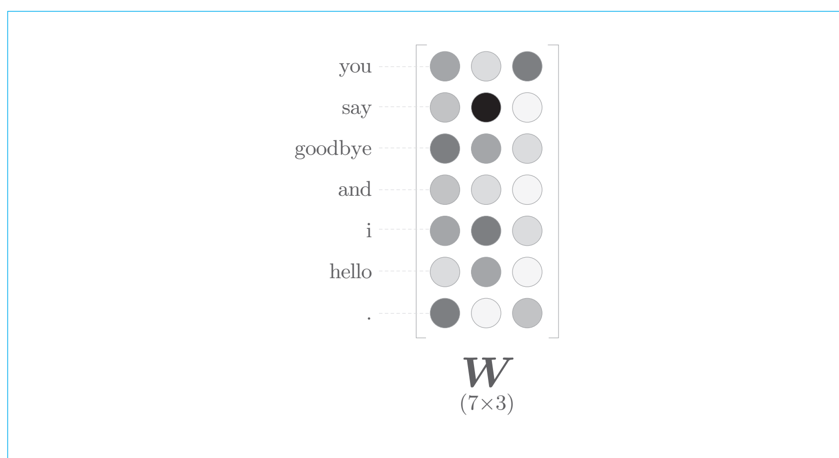


图 3-10　权重的各行对应各个单词的分布式表示

如图 3-10 所示，权重 $\boldsymbol{W}_{\text{in}}$ 的各行保存着各个单词的分布式表示。通过反复学习，不断更新各个单词的分布式表示，以正确地从上下文预测出应当出现的单词。令人惊讶的是，如此获得的向量很好地对单词含义进行了编码。这就是 word2vec 的全貌。



中间层的神经元数量比输入层少这一点很重要。中间层需要将预测单词所需的信息压缩保存，从而产生密集的向量表示。这时，中间层被写入了我们人类无法解读的代码，这相当于“编码”工作。而从中间层的信息获得期望结果的过程则称为“解码”。这一过程将被编码的信息复原为我们可以理解的形式。

到目前为止，我们从神经元视角图示了 CBOW 模型。下面，我们从层视角图示 CBOW 模型。这样一来，这个神经网络的结构如图 3-11 所示。

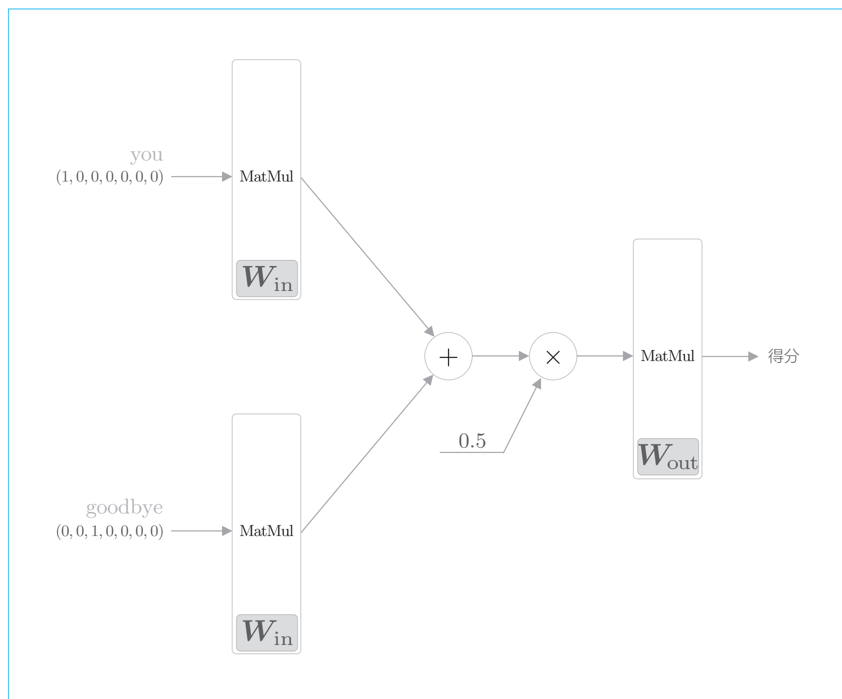


图3-11　层视角下的CBOW模型的网络结构：MatMul层中使用的权重（W_{in}、W_{out}）画在各自的层中

如图 3-11 所示，CBOW 模型一开始有两个 MatMul 层，这两个层的输出被加在一起。然后，对这个相加后得到的值乘以 0.5 求平均，可以得到中间层的神经元。最后，将另一个 MatMul 层应用于中间层的神经元，输出得分。



不使用偏置的全连接层的处理由MatMul层的正向传播代理。这个层在内部计算矩阵乘积。

参考图 3-11，我们来实现 CBOW 模型的推理（即求得分的过程），具体实现如下所示（✏ ch03/cbow_predict.py）。

```
import sys
sys.path.append('..')
import numpy as np
from common.layers import MatMul


# 样本的上下文数据
c0 = np.array([[1, 0, 0, 0, 0, 0, 0]])
c1 = np.array([[0, 0, 1, 0, 0, 0, 0]])

# 权重的初始值
W_in = np.random.randn(7, 3)
W_out = np.random.randn(3, 7)

# 生成层
in_layer0 = MatMul(W_in)
in_layer1 = MatMul(W_in)
out_layer = MatMul(W_out)

# 正向传播
h0 = in_layer0.forward(c0)
h1 = in_layer1.forward(c1)
h = 0.5 * (h0 + h1)
s = out_layer.forward(h)

print(s)
# [[ 0.30916255  0.45060817 -0.77308656  0.22054131  0.15037278
#   -0.93659277 -0.59612048]]
```

这里，我们首先将必要的权重（W_in 和 W_out）初始化。然后，生成与上下文单词数量等量（这里是两个）的处理输入层的 MatMul 层，输出侧仅生成一个 MatMul 层。需要注意的是，输入侧的 MatMul 层共享权重 W_in。

之后，输入侧的 MatMul 层（in_layer0 和 in_layer1）调用 forward() 方法，计算中间数据，并通过输出侧的 MatMul 层（out_layer）计算各个单词的得分。

以上就是 CBOW 模型的推理过程。这里我们见到的 CBOW 模型是没有使用激活函数的简单的网络结构。除了多个输入层共享权重外，并没有什么难点。接下来，我们继续看一下 CBOW 模型的学习。

3.2.2 CBOW模型的学习

到目前为止，我们介绍的CBOW模型在输出层输出了各个单词的得分。通过对这些得分应用 Softmax 函数，可以获得概率（图 3-12）。这个概率表示哪个单词会出现在给定的上下文（周围单词）中间。

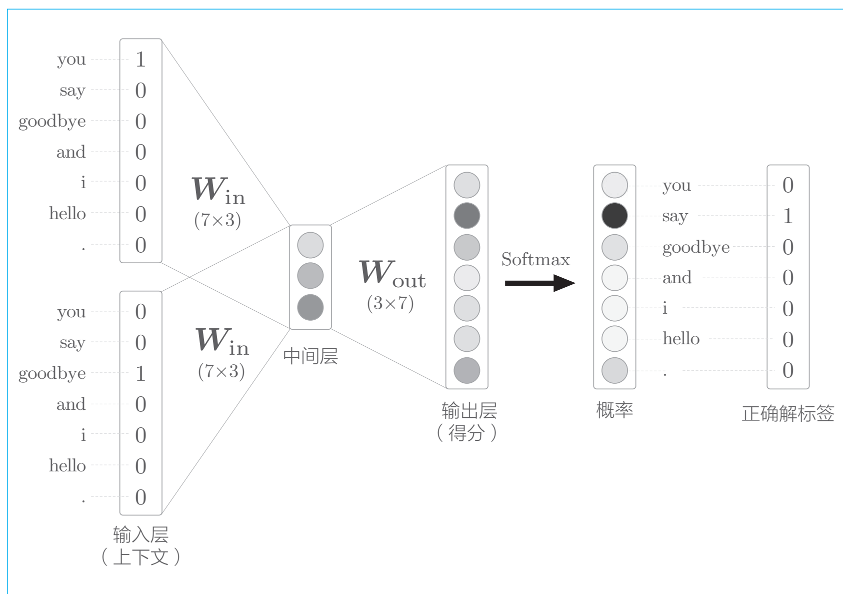


图3-12 CBOW模型的示例（节点值的大小用灰度表示）

在图 3-12 所示的例子中，上下文是 you 和 goodbye，正确解标签（神经网络应该预测出的单词）是 say。这时，如果网络具有“良好的权重”，那么在表示概率的神经元中，对应正确解的神经元的得分应该更高。

CBOW 模型的学习就是调整权重，以使预测准确。其结果是，权重 W_{in}（确切地说是 W_{in} 和 W_{out} 两者）学习到蕴含单词出现模式的向量。根

据过去的实验，CBOW 模型（和 skip-gram 模型）得到的单词的分布式表示，特别是使用维基百科等大规模语料库学习到的单词的分布式表示，在单词的含义和语法上符合我们直觉的案例有很多。



CBOW 模型只是学习语料库中单词的出现模式。如果语料库不一样，学习到的单词的分布式表示也不一样。比如，只使用“体育”相关的文章得到的单词的分布式表示，和只使用“音乐”相关的文章得到的单词的分布式表示将有很大不同。

现在，我们来考虑一下上述神经网络的学习。其实很简单，这里我们处理的模型是一个进行多类别分类的神经网络。因此，对其进行学习只是使用一下 Softmax 函数和交叉熵误差。首先，使用 Softmax 函数将得分转化为概率，再求这些概率和监督标签之间的交叉熵误差，并将其作为损失进行学习，这一过程可以用图 3-13 表示。

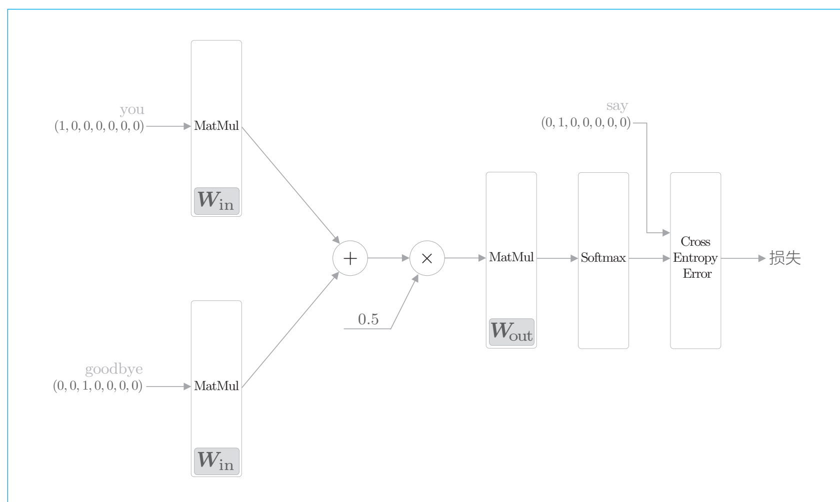


图3-13　学习时的CBOW模型的网络结构

如图 3-13 所示，只需向上一节介绍的进行推理的 CBOW 模型加上 Softmax 层和 Cross Entropy Error 层，就可以得到损失。这就是 CBOW

模型计算损失的流程，对应于神经网络的正向传播。

虽然图 3-13 中使用了 Softmax 层和 Cross Entropy Error 层，但是我们将这两个层实现为了一个 Softmax with Loss 层。因此，我们接下来要实现的网络实际上如图 3-14 所示。

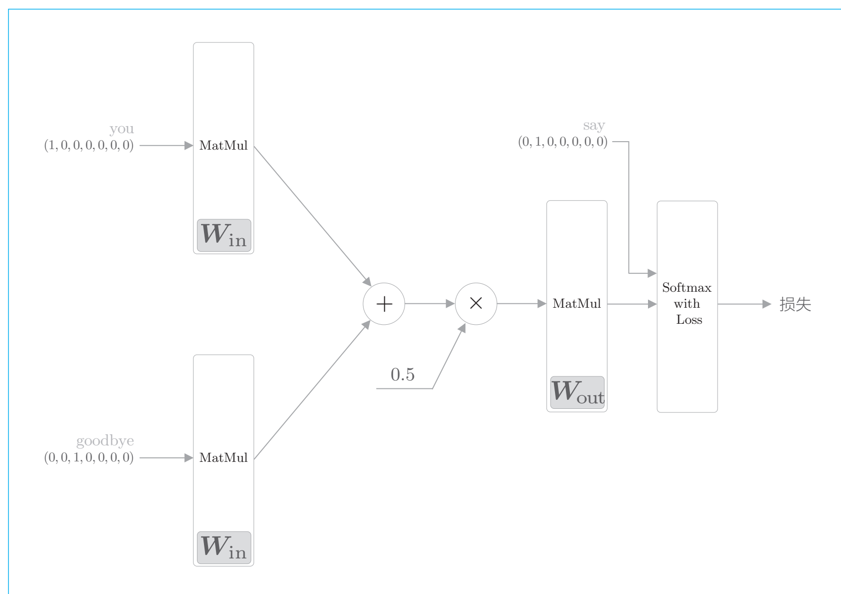


图3-14 将Softmax层和Cross Entropy Error层统一为Softmax with Loss层

3.2.3 word2vec的权重和分布式表示

如前所述，word2vec 中使用的网络有两个权重，分别是输入侧的全连接层的权重（$\boldsymbol{W}_{\text{in}}$）和输出侧的全连接层的权重（$\boldsymbol{W}_{\text{out}}$）。一般而言，输入侧的权重 $\boldsymbol{W}_{\text{in}}$ 的每一行对应于各个单词的分布式表示。另外，输出侧的权重 $\boldsymbol{W}_{\text{out}}$ 也同样保存了对单词含义进行了编码的向量。只是，如图 3-15 所示，输出侧的权重在列方向上保存了各个单词的分布式表示。

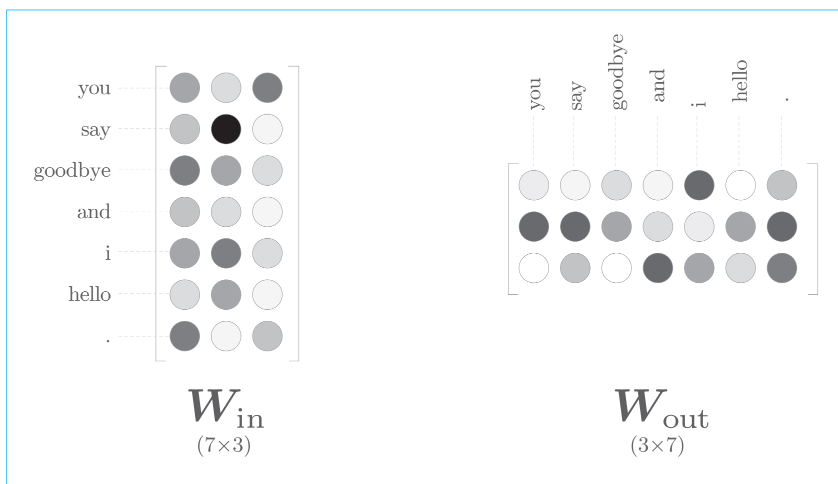


图3-15　输入侧和输出侧的权重都可以被视为单词的分布式表示

那么，我们最终应该使用哪个权重作为单词的分布式表示呢？这里有三个选项。

A. 只使用输入侧的权重
B. 只使用输出侧的权重
C. 同时使用两个权重

方案 A 和方案 B 只使用其中一个权重。而在采用方案 C 的情况下，根据如何组合这两个权重，存在多种方式，其中一个方式就是简单地将这两个权重相加。

就 word2vec（特别是 skip-gram 模型）而言，最受欢迎的是方案 A。许多研究中也都仅使用输入侧的权重 $\boldsymbol{W}_{in}$ 作为最终的单词的分布式表示。遵循这一思路，我们也使用 $\boldsymbol{W}_{in}$ 作为单词的分布式表示。



文献 [38] 通过实验证明了 word2vec 的 skip-gram 模型中 $\boldsymbol{W}_{in}$ 的有效性。另外，在与 word2vec 相似的 GloVe[27] 方法中，通过将两个权重相加，也获得了良好的结果。

3.3 学习数据的准备

在开始 word2vec 的学习之前，我们先来准备学习用的数据。这里我们仍以“You say goodbye and I say hello.”这个只有一句话的语料库为例进行说明。

3.3.1 上下文和目标词

word2vec 中使用的神经网络的输入是上下文，它的正确解标签是被这些上下文包围在中间的单词，即目标词。也就是说，我们要做的事情是，当向神经网络输入上下文时，使目标词出现的概率高（为了达成这一目标而进行学习）。

下面我们就从语料库生成上下文和目标词，如图 3-16 所示。

corpus	contexts	target
you say goodbye and i say hello .	you, goodbye	say
you say goodbye and i say hello .	say, and	goodbye
you say goodbye and i say hello .	goodbye, i	and
you say goodbye and i say hello .	and, say	i
you say goodbye and i say hello .	i, hello	say
you say goodbye and i say hello .	say, .	hello

图 3-16 从语料库生成上下文和目标词

在图 3-16 中，将语料库中的目标单词作为目标词，将其周围的单词作为上下文提取出来。我们对语料库中的所有单词都执行该操作（两端的单词除外），可以得到图 3-16 右侧的 `contexts`（上下文）和 `target`（目标词）。`contexts` 的各行成为神经网络的输入，`target` 的各行成为正确解标签（要预测出的单词）。另外，在各笔样本数据中，上下文有多个单词（这个例子中有两个），而目标词则只有一个，因此只有上下文写成了复数形式 `contexts`。

现在，我们来实现从语料库生成上下文和目标词的函数。在此之前，我们先复习一下上一章的内容。首先，将语料库的文本转化成单词 ID。这需要使用第 2 章实现的 `preprocess()` 函数。

```
import sys
sys.path.append('..')
from common.util import preprocess

text = 'You say goodbye and I say hello.'
corpus, word_to_id, id_to_word = preprocess(text)
print(corpus)
# [0 1 2 3 4 1 5 6]

print(id_to_word)
# {0: 'you', 1: 'say', 2: 'goodbye', 3: 'and', 4: 'i', 5: 'hello', 6:
↪  '.'}
```

然后，从单词 ID 列表 `corpus` 生成 `contexts` 和 `target`。具体来说，如图 3-17 所示，实现一个当给定 `corpus` 时返回 `contexts` 和 `target` 的函数。

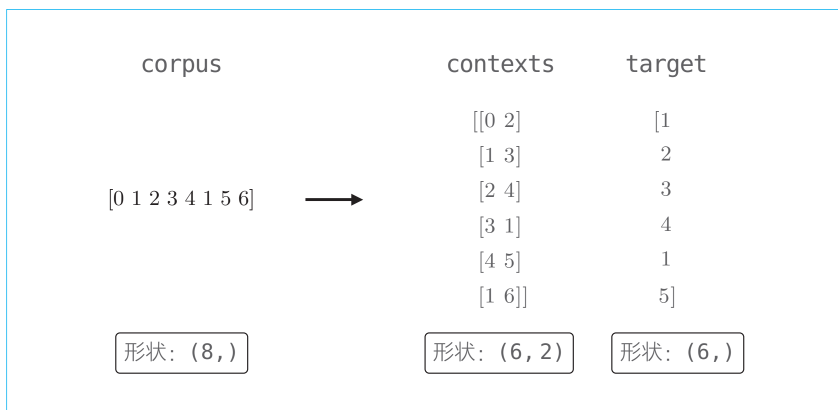


图 3-17　从单词 ID 列表 corpus 生成 contexts 和 target 的例子（上下文的窗口大小为 1）

如图 3-17 所示，`contexts` 是二维数组。此时，`contexts` 的第 0 维保存的是各个上下文数据。具体来说，`contexts[0]` 保存的是第 0 个上下文，

context[1] 保存的是第 1 个上下文……同样地，就目标词而言，target[0] 保存的是第 0 个目标词，target[1] 保存的是第 1 个目标词……

现在，我们来实现这个生成上下文和目标词的函数，这里将其称为 create_contexts_target(corpus, window_size)（✏ common/util.py）。

```
def create_contexts_target(corpus, window_size=1):
    target = corpus[window_size:-window_size]
    contexts = []

    for idx in range(window_size, len(corpus)-window_size):
        cs = []
        for t in range(-window_size, window_size + 1):
            if t == 0:
                continue
            cs.append(corpus[idx + t])
        contexts.append(cs)

    return np.array(contexts), np.array(target)
```

这个函数有两个参数：一个是单词 ID 列表（corpus）；另一个是上下文的窗口大小（window_size）。另外，函数返回的是 NumPy 多维数组格式的上下文和目标词。现在，我们来实际使用一下这个函数，接着刚才的实现，代码如下所示。

```
contexts, target = create_contexts_target(corpus, window_size=1)

print(contexts)
# [[0 2]
#  [1 3]
#  [2 4]
#  [3 1]
#  [4 5]
#  [1 6]]

print(target)
# [1 2 3 4 1 5]
```

这样就从语料库生成了上下文和目标词，后面只需将它们赋给 CBOW 模型即可。不过，因为这些上下文和目标词的元素还是单词 ID，所以还需要将它们转化为 one-hot 表示。

3.3.2　转化为 one-hot 表示

下面，我们将上下文和目标词转化为 one-hot 表示，如图 3-18 所示。

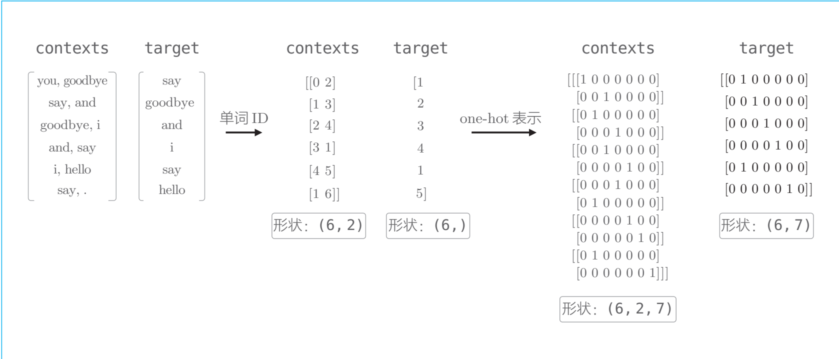


图 3-18　将上下文和目标词转化为 one-hot 表示的例子

如图 3-18 所示，上下文和目标词从单词 ID 转化为了 one-hot 表示。这里需要注意各个多维数组的形状。在上面的例子中，使用单词 ID 时的 `contexts` 的形状是 `(6,2)`，将其转化为 one-hot 表示后，形状变为 `(6,2,7)`。

本书提供了 `convert_one_hot()` 函数以将单词 ID 转化为 one-hot 表示。这个函数的实现不再说明，内容很简单，代码在 `common/util.py` 中。该函数的参数是单词 ID 列表和词汇个数。我们再把到目前为止的数据预处理总结一下，如下所示。

```
import sys
sys.path.append('..')
from common.util import preprocess, create_contexts_target,
 ↪   convert_one_hot

text = 'You say goodbye and I say hello.'
```

```
corpus, word_to_id, id_to_word = preprocess(text)

contexts, target = create_contexts_target(corpus, window_size=1)

vocab_size = len(word_to_id)
target = convert_one_hot(target, vocab_size)
contexts = convert_one_hot(contexts, vocab_size)
```

至此，学习数据的准备就完成了，下面我们来讨论最重要的 CBOW 模型的实现。

3.4 CBOW模型的实现

现在，我们来实现 CBOW 模型。这里要实现的神经网络如图 3-19 所示。

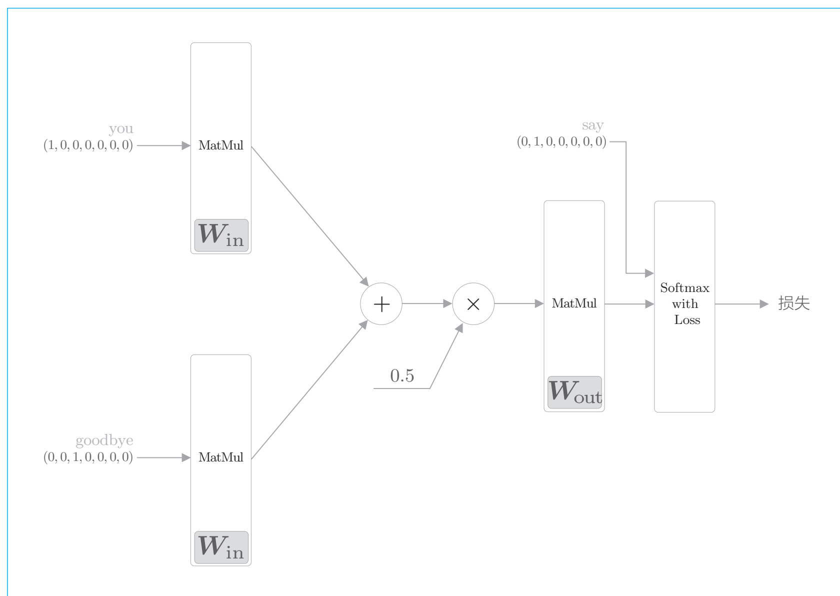


图3-19 CBOW模型的网络结构

我们将图 3-19 中的神经网络实现为 SimpleCBOW 类（下一章将实现对其进行了改进的 CBOW 类）。首先，让我们看一下 SimpleCBOW 类的初始化方法（✏ ch03/simple_cbow.py）。

```
import sys
sys.path.append('..')
import numpy as np
from common.layers import MatMul, SoftmaxWithLoss

class SimpleCBOW:
    def __init__(self, vocab_size, hidden_size):
        V, H = vocab_size, hidden_size

        # 初始化权重
        W_in = 0.01 * np.random.randn(V, H).astype('f')
        W_out = 0.01 * np.random.randn(H, V).astype('f')

        # 生成层
        self.in_layer0 = MatMul(W_in)
        self.in_layer1 = MatMul(W_in)
        self.out_layer = MatMul(W_out)
        self.loss_layer = SoftmaxWithLoss()

        # 将所有的权重和梯度整理到列表中
        layers = [self.in_layer0, self.in_layer1, self.out_layer]
        self.params, self.grads = [], []
        for layer in layers:
            self.params += layer.params
            self.grads += layer.grads

        # 将单词的分布式表示设置为成员变量
        self.word_vecs = W_in
```

这里，初始化方法的参数包括词汇个数 vocab_size 和中间层的神经元个数 hidden_size。关于权重的初始化，首先我们生成两个权重（W_in 和 W_out），并用一些小的随机值初始化这两个权重。此外，我们指定 NumPy 数组的数据类型为 astype('f')，这样一来，初始化将使用 32 位的浮点数。

接着，我们创建必要的层。首先，生成两个输入侧的 MatMul 层、一个输出侧的 MatMul 层，以及一个 Softmax with Loss 层。这里，用来处理输入侧上下文的 MatMul 层的数量与上下文的单词数量相同（本例中是两个）。另外，我们使用相同的权重来初始化 MatMul 层。

最后，将该神经网络中使用的权重参数和梯度分别保存在列表类型的成员变量 params 和 grads 中。



这里，多个层共享相同的权重。因此，params 列表中存在多个相同的权重。但是，在 params 列表中存在多个相同的权重的情况下，Adam、Momentum 等优化器的运行会变得不符合预期（至少就我们的代码而言）。为此，在 Trainer 类的内部，在更新参数时会进行简单的去重操作。关于这一点，这里省略说明，感兴趣的读者可以参考 common/trainer.py 的 remove_duplicate(params, grads)。

接下来，我们来实现神经网络的正向传播 forward() 函数。这个函数接收参数 contexts 和 target，并返回损失（loss）。

```
def forward(self, contexts, target):
    h0 = self.in_layer0.forward(contexts[:, 0])
    h1 = self.in_layer1.forward(contexts[:, 1])
    h = (h0 + h1) * 0.5
    score = self.out_layer.forward(h)
    loss = self.loss_layer.forward(score, target)
    return loss
```

这里，我们假定参数 contexts 是一个三维 NumPy 数组，即上一节图 3-18 的例子中 (6,2,7) 的形状，其中第 0 维的元素个数是 mini-batch 的数量，第 1 维的元素个数是上下文的窗口大小，第 2 维表示 one-hot 向量。此外，target 是 (6,7) 这样的二维形状。

最后，我们实现反向传播 backward()。这个反向传播的计算图如图 3-20 所示。

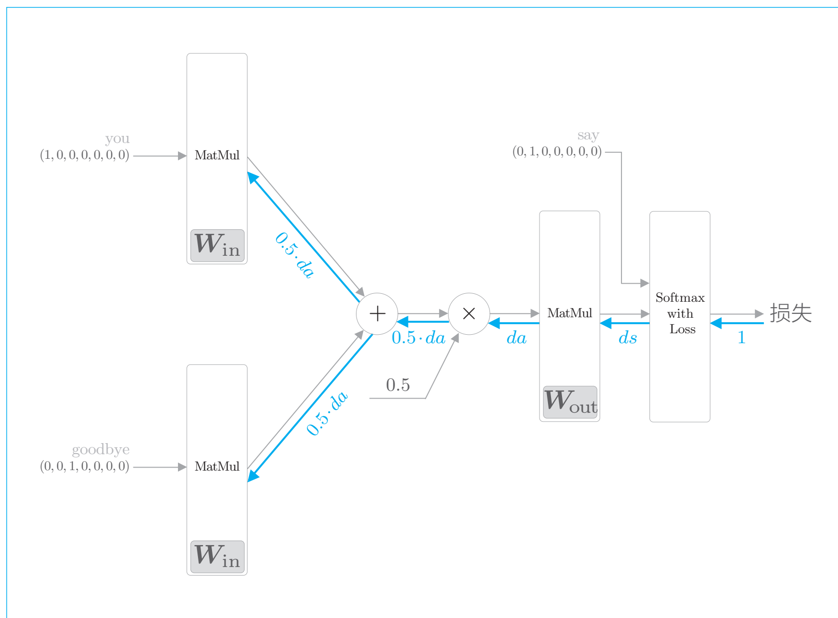


图3-20　CBOW模型的反向传播：蓝色的粗线表示反向传播的路线

神经网络的反向传播在与正向传播相反的方向上传播梯度。这个反向传播从 1 出发，并将其传向 Softmax with Loss 层。然后，将 Softmax with Loss 层的反向传播的输出 ds 传到输出侧的 MatMul 层。

之后就是“+”和“×”运算的反向传播。“×”的反向传播将正向传播时的输入值“交换”后乘以梯度。“+”的反向传播则将梯度“原样”传播。我们按照图 3-20 来实现反向传播。

```
def backward(self, dout=1):
    ds = self.loss_layer.backward(dout)
    da = self.out_layer.backward(ds)
    da *= 0.5
    self.in_layer1.backward(da)
    self.in_layer0.backward(da)
    return None
```

至此，反向传播的实现就结束了。我们已经将各个权重参数的梯度

保存在了成员变量 grads 中。因此，通过先调用 forward() 函数，再调用 backward() 函数，grads 列表中的梯度被更新。下面，我们继续看一下 SimpleCBOW 类的学习。

学习的实现

CBOW 模型的学习和一般的神经网络的学习完全相同。首先，给神经网络准备好学习数据。然后，求梯度，并逐步更新权重参数。这里，我们使用第 1 章介绍的 Trainer 类来执行学习过程，学习的源代码如下所示（✏ ch03/train.py）。

```
import sys
sys.path.append('..')
from common.trainer import Trainer
from common.optimizer import Adam
from simple_cbow import SimpleCBOW
from common.util import preprocess, create_contexts_target,
↪   convert_one_hot


window_size = 1
hidden_size = 5
batch_size = 3
max_epoch = 1000

text = 'You say goodbye and I say hello.'
corpus, word_to_id, id_to_word = preprocess(text)

vocab_size = len(word_to_id)
contexts, target = create_contexts_target(corpus, window_size)
target = convert_one_hot(target, vocab_size)
contexts = convert_one_hot(contexts, vocab_size)

model = SimpleCBOW(vocab_size, hidden_size)
optimizer = Adam()
trainer = Trainer(model, optimizer)
```

```
trainer.fit(contexts, target, max_epoch, batch_size)
trainer.plot()
```

`common/optimizer.py` 中实现了 SGD、AdaGrad 等多个著名的参数更新方法。这里，我们选择 Adam 算法。如第 1 章所述，`Trainer` 类会执行神经网络的学习过程，包括从学习数据中选出 mini-batch 给神经网络以算出梯度，并将这个梯度给优化器以更新权重参数等一系列操作。



之后，我们也使用 `Trainer` 类进行神经网络的学习。使用 `Trainer` 类，可以理清容易变复杂的学习代码。

运行上面的代码，结果如图 3-21 所示。

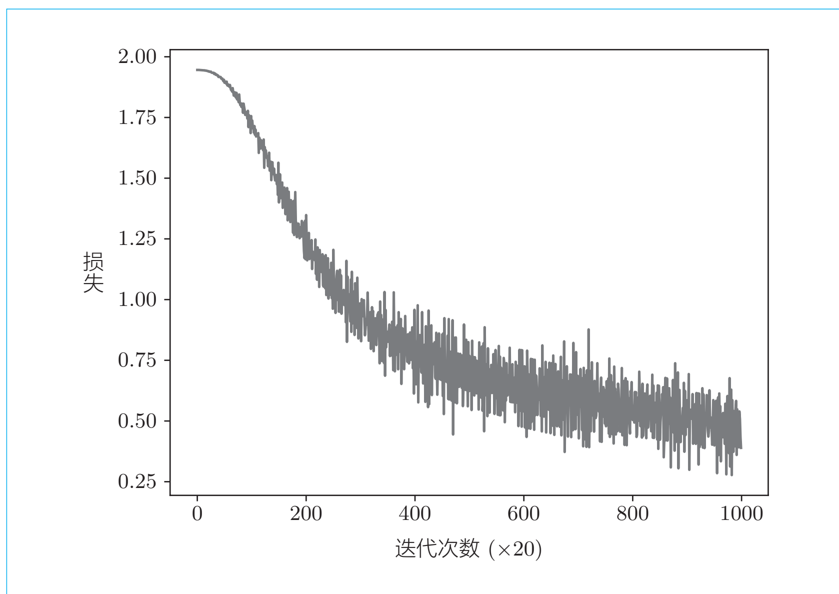


图 3-21 用图形表示学习过程（横轴表示学习的迭代次数，纵轴表示损失）

如图 3-21 所示，通过不断学习，损失在减小，看起来学习进行得一切正常。我们来看一下学习结束后的权重参数。这里，我们取出输入侧的

MatMul 层的权重，实际确认一下它的内容。因为输入侧的 MatMul 层的权重已经赋值给了成员变量 word_vecs，所以接着上面的代码，我们追加下面的代码。

```
word_vecs = model.word_vecs
for word_id, word in id_to_word.items():
    print(word, word_vecs[word_id])
```

这里，使用 word_vecs 这个变量保存权重。word_vecs 的各行保存了对应的单词 ID 的分布式表示。实际运行一下，可以得到下述结果。

```
you [-0.9031807  -1.0374491  -1.4682057  -1.3216232   0.93127245]
say [ 1.2172916   1.2620505  -0.07845993  0.07709391 -1.2389531 ]
goodbye [-1.0834033  -0.8826921  -0.33428606 -0.5720131   1.0488235 ]
and [ 1.0244362  1.0160093 -1.6284224 -1.6400533 -1.0564581]
i [-1.0642933  -0.9162385  -0.31357735 -0.5730831   1.041875   ]
hello [-0.9018145 -1.035476  -1.4629668 -1.3058501  0.9280102]
. [ 1.0985303  1.1642815  1.4365371  1.3974973 -1.0714306]
```

我们终于将单词表示为了密集向量！这就是单词的分布式表示。我们有理由相信，这样的分布式表示能够很好地捕获单词含义。

不过，遗憾的是，这里使用的小型语料库并没有给出很好的结果。当然，主要原因是语料库太小了。如果换成更大、更实用的语料库，相信会获得更好的结果。但是，这样在处理速度方面又会出现新的问题，这是因为当前这个 CBOW 模型的实现在处理效率方面存在几个问题。下一章我们将改进这个简单的 CBOW 模型，实现一个“真正的”CBOW 模型。

3.5 word2vec的补充说明

至此，我们详细探讨了 word2vec 的 CBOW 模型。接下来，我们将对 word2vec 补充说明几个非常重要的话题。首先，我们从概率的角度，再来看一下 CBOW 模型。

3.5.1 CBOW模型和概率

首先简单说明一下概率的表示方法。本书中将概率记为 $P(\cdot)$，比如事件 A 发生的概率记为 $P(A)$。联合概率记为 $P(A,B)$，表示事件 A 和事件 B 同时发生的概率。

后验概率记为 $P(A|B)$，字面意思是“事件发生后的概率”。从另一个角度来看，也可以解释为“在给定事件 B（的信息）时事件 A 发生的概率”。

下面，我们用概率的表示方法来描述 CBOW 模型。CBOW 模型进行的处理是，当给定某个上下文时，输出目标词的概率。这里，我们使用包含单词 $w_1, w_2, \cdots, w_T$ 的语料库。如图 3-22 所示，对第 t 个单词，考虑窗口大小为 1 的上下文。

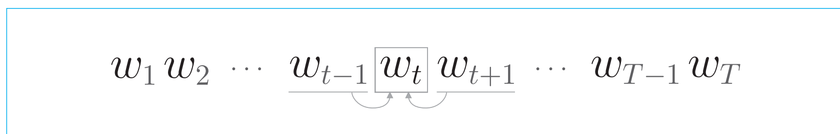


图 3-22 word2vec的CBOW模型：从上下文的单词预测目标词

下面，我们用数学式来表示当给定上下文 w_{t-1} 和 w_{t+1} 时目标词为 w_t 的概率。使用后验概率，有式 (3.1)：

$$P(w_t|w_{t-1}, w_{t+1}) \tag{3.1}$$

式 (3.1) 表示“在 w_{t-1} 和 w_{t+1} 发生后，w_t 发生的概率”，也可以解释为“当给定 w_{t-1} 和 w_{t+1} 时，w_t 发生的概率”。也就是说，CBOW 模型可以建模为式 (3.1)。

这里，使用式 (3.1) 可以简洁地表示 CBOW 模型的损失函数。我们把第 1 章介绍的交叉熵误差函数（式 (1.7)）套用在这里。式 (1.7) 是 $L = -\sum_k t_k \log y_k$，其中，y_k 表示第 k 个事件发生的概率。t_k 是监督标签，它是 one-hot 向量的元素。这里需要注意的是，“w_t 发生”这一事件是正确解，它对应的 one-hot 向量的元素是 1，其他元素都是 0（也就是说，当 w_t

之外的事件发生时，对应的 one-hot 向量的元素均为 0）。考虑到这一点，可以推导出下式：

$$L = -\log P(w_t|w_{t-1}, w_{t+1}) \tag{3.2}$$

CBOW 模型的损失函数只是对式 (3.1) 的概率取 log，并加上负号。顺便提一下，这也称为负对数似然（negative log likelihood）。式 (3.2) 是一笔样本数据的损失函数。如果将其扩展到整个语料库，则损失函数可以写为：

$$L = -\frac{1}{T}\sum_{t=1}^{T} \log P(w_t|w_{t-1}, w_{t+1}) \tag{3.3}$$

CBOW 模型学习的任务就是让式 (3.3) 表示的损失函数尽可能地小。那时的权重参数就是我们想要的单词的分布式表示。这里，我们只考虑了窗口大小为 1 的情况，不过其他的窗口大小（或者窗口大小为 m 的一般情况）也很容易用数学式表示。

3.5.2 skip-gram模型

如前所述，word2vec 有两个模型：一个是我们已经讨论过的 CBOW 模型；另一个是被称为 skip-gram 的模型。skip-gram 是反转了 CBOW 模型处理的上下文和目标词的模型。举例来说，两者要解决的问题如图 3-23 所示。

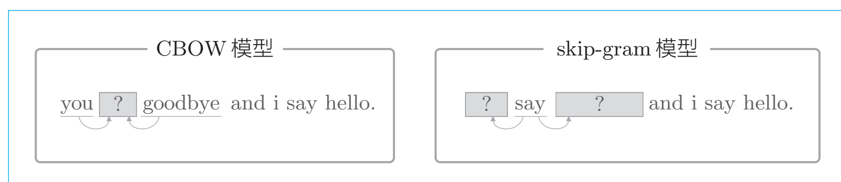


图 3-23 CBOW 模型和 skip-gram 模型处理的问题

如图 3-23 所示，CBOW 模型从上下文的多个单词预测中间的单词（目

标词），而 skip-gram 模型则从中间的单词（目标词）预测周围的多个单词（上下文）。此时，skip-gram 模型的网络结构如图 3-24 所示。

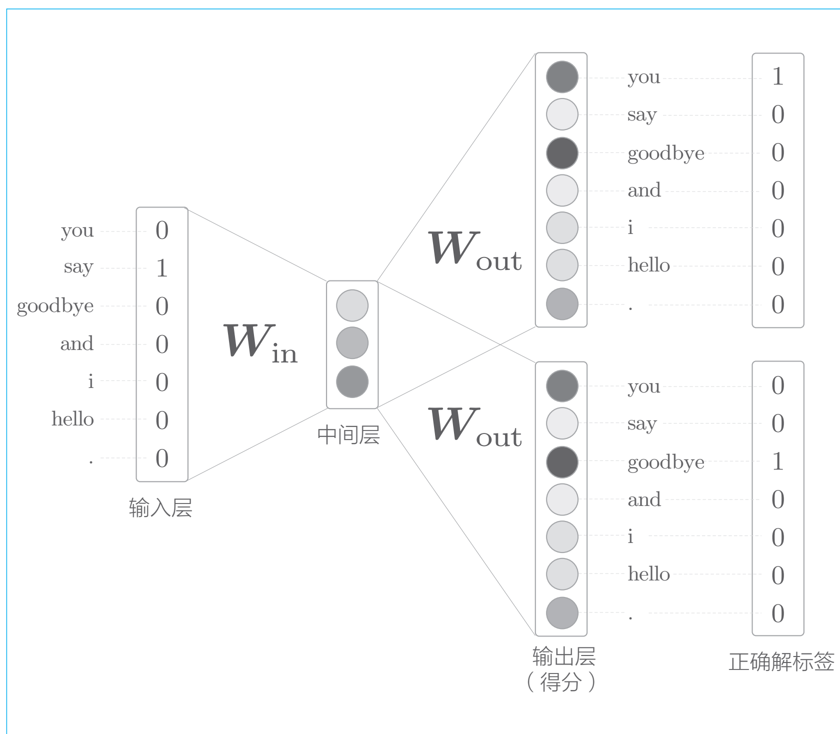


图3-24 skip-gram模型的例子

由图 3-24 可知，skip-gram 模型的输入层只有一个，输出层的数量则与上下文的单词个数相等。因此，首先要分别求出各个输出层的损失（通过 Softmax with Loss 层等），然后将它们加起来作为最后的损失。

现在，我们使用概率的表示方法来表示 skip-gram 模型。我们来考虑根据中间单词（目标词）w_t 预测上下文 w_{t-1} 和 w_{t+1} 的情况。此时，skip-gram 可以建模为式 (3.4)：

$$P(w_{t-1}, w_{t+1} | w_t) \tag{3.4}$$

式 (3.4) 表示“当给定 w_t 时，w_{t-1} 和 w_{t+1} 同时发生的概率”。这里，在 skip-gram 模型中，假定上下文的单词之间没有相关性（正确地说是假定“条件独立”），将式 (3.4) 如下进行分解：

$$P(w_{t-1}, w_{t+1}|w_t) = P(w_{t-1}|w_t)\,P(w_{t+1}|w_t) \tag{3.5}$$

通过将式 (3.5) 代入交叉熵误差函数，可以推导出 skip-gram 模型的损失函数：

$$\begin{aligned} L &= -\log P(w_{t-1}, w_{t+1}|w_t) \\ &= -\log P(w_{t-1}|w_t)P(w_{t+1}|w_t) \\ &= -(\log P(w_{t-1}|w_t) + \log P(w_{t+1}|w_t)) \end{aligned} \tag{3.6}$$

这里利用了对数的性质 $\log xy = \log x + \log y$。如式 (3.6) 所示，skip-gram 模型的损失函数先分别求出各个上下文对应的损失，然后将它们加在一起。式 (3.6) 是一笔样本数据的损失函数。如果扩展到整个语料库，则 skip-gram 模型的损失函数可以表示为式 (3.7)：

$$L = -\frac{1}{T}\sum_{t=1}^{T}(\log P(w_{t-1}|w_t) + \log P(w_{t+1}|w_t)) \tag{3.7}$$

比较式 (3.7) 和 CBOW 模型的式 (3.3)，差异是非常明显的。因为 skip-gram 模型的预测次数和上下文单词数量一样多，所以它的损失函数需要求各个上下文单词对应的损失的总和。而 CBOW 模型只需要求目标词的损失。以上就是对 skip-gram 模型的介绍。

那么，我们应该使用 CBOW 模型和 skip-gram 模型中的哪一个呢？答案应该是 skip-gram 模型。这是因为，从单词的分布式表示的准确度来看，在大多数情况下，skip-grm 模型的结果更好。特别是随着语料库规模的增大，在低频词和类推问题的性能方面，skip-gram 模型往往会有更好的表现（单词的分布式表示的评价方法会在 4.4.2 节说明）。此外，就学习速度而言，CBOW 模型比 skip-gram 模型要快。这是因为 skip-gram 模型需要根据上

下文数量计算相应个数的损失，计算成本变大。



skip-gram 模型根据一个单词预测其周围的单词，这是一个非常难的问题。假如我们来解决图 3-23 中的问题，此时，对于 CBOW 模型的问题，我们很容易回答“say”。但是，对于 skip-gram 模型的问题，则存在许多候选。因此，可以说 skip-gram 模型要解决的是更难的问题。经过这个更难的问题的锻炼，skip-gram 模型能提供更好的单词的分布式表示。

理解了 CBOW 模型的实现，在实现 skip-gram 模型时应该就不存在什么难点了。因此，这里就不再介绍 skip-gram 模型的实现。感兴趣的读者可以参考 `ch03/simple_skip_gram.py`。

3.5.3 基于计数与基于推理

到目前为止，我们已经了解了基于计数的方法和基于推理的方法（特别是 word2vec）。两种方法在学习机制上存在显著差异：基于计数的方法通过对整个语料库的统计数据进行一次学习来获得单词的分布式表示，而基于推理的方法则反复观察语料库的一部分数据进行学习（mini-batch 学习）。这里，我们就其他方面来对比一下这两种方法。

首先，我们考虑需要向词汇表添加新词并更新单词的分布式表示的场景。此时，基于计数的方法需要从头开始计算。即便是想稍微修改一下单词的分布式表示，也需要重新完成生成共现矩阵、进行 SVD 等一系列操作。相反，基于推理的方法（word2vec）允许参数的增量学习。具体来说，可以将之前学习到的权重作为下一次学习的初始值，在不损失之前学习到的经验的情况下，高效地更新单词的分布式表示。在这方面，基于推理的方法（word2vec）具有优势。

其次，两种方法得到的单词的分布式表示的性质和准确度有什么差异呢？就分布式表示的性质而言，基于计数的方法主要是编码单词的相似性，而 word2vec（特别是 skip-gram 模型）除了单词的相似性以外，还能

理解更复杂的单词之间的模式。关于这一点，word2vec 因能解开“king − man + woman = queen”这样的类推问题而知名（关于类推问题，我们将在 4.4.2 节说明）。

这里有一个常见的误解，那就是基于推理的方法在准确度方面优于基于计数的方法。实际上，有研究表明，就单词相似性的定量评价而言，基于推理的方法和基于计数的方法难分上下 [25]。



2014 年发表的题为“Don't count, predict!”(不要计数，要预测!) 的论文 [24] 系统地比较了基于计数的方法和基于推理的方法，并给出了基于推理的方法在准确度上始终更好的结论。但是，之后又有其他的论文 [25] 提出，就单词的相似性而言，结论高度依赖于超参数，基于计数的方法和基于推理的方法难分胜负。

另外一个重要的事实是，基于推理的方法和基于计数的方法存在关联性。具体地说，使用了 skip-gram 和下一章介绍的 Negative Sampling 的模型被证明与对整个语料库的共现矩阵（实际上会对矩阵进行一定的修改）进行特殊矩阵分解的方法具有相同的作用 [26]。换句话说，这两个方法论（在某些条件下）是“相通”的。

此外，在 word2vec 之后，有研究人员提出了 GloVe 方法 [27]。GloVe 方法融合了基于推理的方法和基于计数的方法。该方法的思想是，将整个语料库的统计数据的信息纳入损失函数，进行 mini-batch 学习（具体请参考论文 [27]）。据此，这两个方法论成功地被融合在了一起。

3.6 小结

托马斯 · 米科洛夫（Tomas Mikolov）在一系列论文[22][23]中提出了word2vec。自论文发表以来，word2vec 受到了许多关注，它的作用也在许多自然语言处理任务中得到了证明。下一章，我们将结合具体的例子来说明 word2vec 的重要性，特别是 word2vec 的迁移学习的作用。

本章我们详细解释了 word2vec 的 CBOW 模型，并对其进行了实现。CBOW 模型基本上是一个 2 层的神经网络，结构非常简单。我们使用 MatMul 层和 Softmax with Loss 层构建了 CBOW 模型，并用一个小规模语料库确认了它的学习过程。遗憾的是，现阶段的 CBOW 模型在处理效率上还存在一些问题。不过，在理解了本章的 CBOW 模型之后，离真正的 word2vec 也就一步之遥了。下一章，我们将改进 CBOW 模型。

本章所学的内容

- 基于推理的方法以预测为目标，同时获得了作为副产物的单词的分布式表示
- word2vec 是基于推理的方法，由简单的 2 层神经网络构成
- word2vec 有 skip-gram 模型和 CBOW 模型
- CBOW 模型从多个单词（上下文）预测 1 个单词（目标词）
- skip-gram 模型反过来从 1 个单词（目标词）预测多个单词（上下文）
- 由于 word2vec 可以进行权重的增量学习，所以能够高效地更新或添加单词的分布式表示

第4章 word2vec的高速化

不要企图无所不知，否则你将一无所知。

——德谟克利特（古希腊哲学家）

上一章我们学习了 word2vec 的机制，并实现了 CBOW 模型。因为 CBOW 模型是一个简单的 2 层神经网络，所以实现起来比较简单。但是，目前的实现存在几个问题，其中最大的问题是，随着语料库中处理的词汇量的增加，计算量也随之增加。实际上，当词汇量达到一定程度之后，上一章的 CBOW 模型的计算就会花费过多的时间。

因此，本章将重点放在 word2vec 的加速上，来改善 word2vec。具体而言，我们将对上一章中简单的 word2vec 进行两点改进：引入名为 Embedding 层的新层，以及引入名为 Negative Sampling 的新损失函数。这样一来，我们就能够完成一个“真正的”word2vec。完成这个真正的 word2vec 后，我们将在 PTB 数据集（一个大小比较实用的语料库）上进行学习，并实际评估所获得的单词的分布式表示的优劣。

4.1 word2vec的改进①

我们先复习一下上一章的内容。在上一章中，我们实现了图 4-1 中的 CBOW 模型。

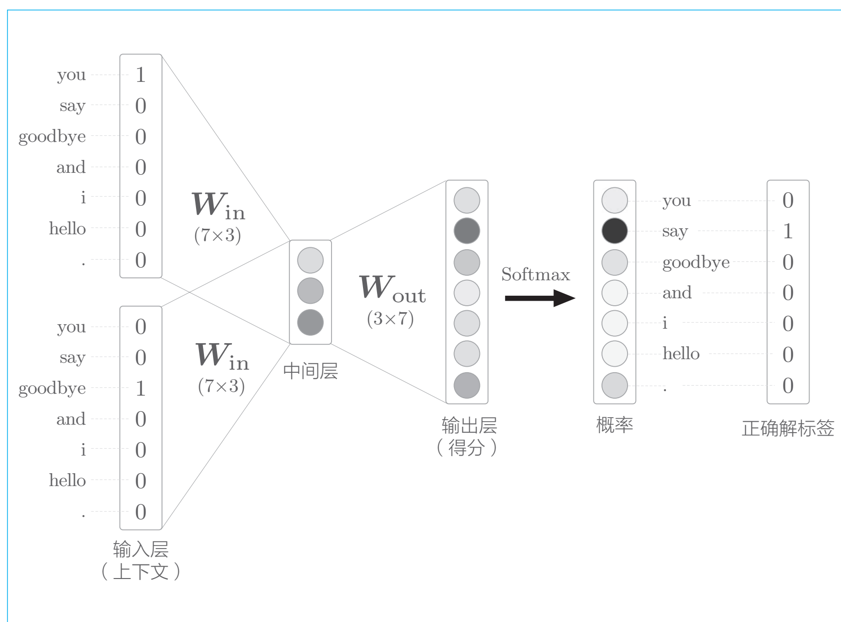


图 4-1　上一章中实现的 CBOW 模型

如图 4-1 所示，上一章的 CBOW 模型接收拥有 2 个单词的上下文，并基于它们预测 1 个单词（目标词）。此时，通过输入层和输入侧权重（$\boldsymbol{W}_{\text{in}}$）之间的矩阵乘积计算中间层，通过中间层和输出侧权重（$\boldsymbol{W}_{\text{out}}$）之间的矩阵乘积计算每个单词的得分。这些得分经过 Softmax 函数转化后，得到每个单词的出现概率。通过将这些概率与正确解标签进行比较（更确切地说，使用交叉熵误差函数），从而计算出损失。



在上一章中，我们限定了上下文的窗口大小为 1。这相当于只将目标词的前一个和后一个单词作为上下文。本章我们将给模型新增一个功能，使之能够处理任意窗口大小的上下文。

图 4-1 中的 CBOW 模型在处理小型语料库时问题不大。实际上，图 4-1 中处理的词汇量一共只有 7 个，这个规模自然毫无问题。不过在处理大规模语料库时，这个模型就存在多个问题了。为了指出这些问题，这里我们考虑

一个例子。假设词汇量有 100 万个，CBOW 模型的中间层神经元有 100 个，此时 word2vec 进行的处理如图 4-2 所示。

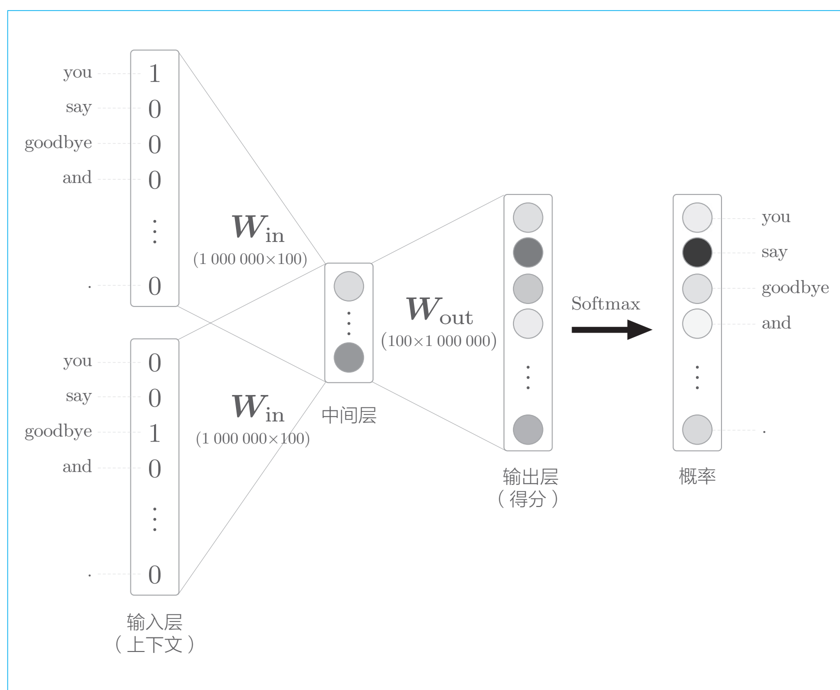


图 4-2　假设词汇量为 100 万个时的 CBOW 模型

如图 4-2 所示，输入层和输出层存在 100 万个神经元。在如此多的神经元的情况下，中间的计算过程需要很长时间。具体来说，以下两个地方的计算会出现瓶颈。

- 输入层的 one-hot 表示和权重矩阵 $\boldsymbol{W}_{\text{in}}$ 的乘积（4.1 节解决）
- 中间层和权重矩阵 $\boldsymbol{W}_{\text{out}}$ 的乘积以及 Softmax 层的计算（4.2 节解决）

第 1 个问题与输入层的 one-hot 表示有关。这是因为我们用 one-hot 表示来处理单词，随着词汇量的增加，one-hot 表示的向量大小也会增加。比如，在词汇量有 100 万个的情况下，仅 one-hot 表示本身就需要占用 100 万

个元素的内存大小。此外，还需要计算 one-hot 表示和权重矩阵 $\boldsymbol{W}_{\text{in}}$ 的乘积，这也要花费大量的计算资源。关于这个问题，我们会在 4.1 节中通过引入新的 Embedding 层来解决。

第 2 个问题是中间层之后的计算。首先，中间层和权重矩阵 $\boldsymbol{W}_{\text{out}}$ 的乘积需要大量的计算。其次，随着词汇量的增加，Softmax 层的计算量也会增加。关于这些问题，我们将在 4.2 节通过引入 Negative Sampling 这一新的损失函数来解决。下面就让我们通过改进来消除这两个瓶颈。



改进前的版本（上一章中的word2vec实现）在`ch03`目录下的`simple_cbow.py`（或者`simple_skip_gram.py`）中。改进后的word2vec版本在`ch04`目录下的`cbow.py`（或者`skip_gram.py`）中。

4.1.1 Embedding层

在上一章的 word2vec 实现中，我们将单词转化为了 one-hot 表示，并将其输入了 MatMul 层，在 MatMul 层中计算了该 one-hot 表示和权重矩阵的乘积。这里，我们来考虑词汇量是 100 万个的情况。假设中间层的神经元个数是 100，则 MatMul 层中的矩阵乘积可画成图 4-3。

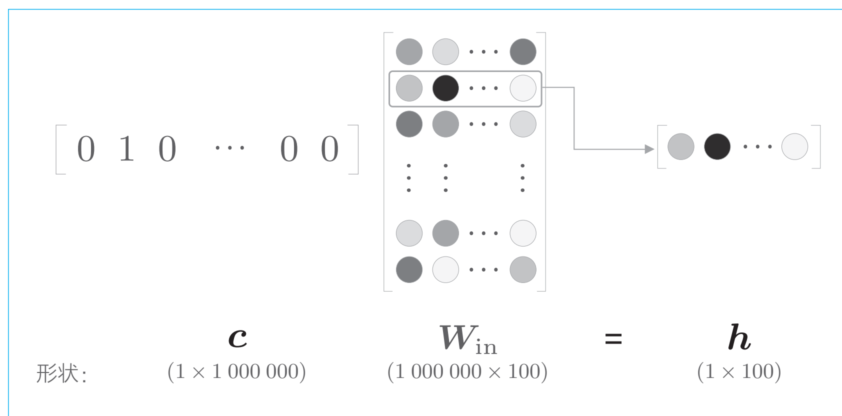


图4-3　one-hot表示的上下文和MatMul层的权重的乘积

如图 4-3 所示，如果语料库的词汇量有 100 万个，则单词的 one-hot 表示的维数也会是 100 万，我们需要计算这个巨大向量和权重矩阵的乘积。但是，图 4-3 中所做的无非是将矩阵的某个特定的行取出来。因此，直觉上将单词转化为 one-hot 向量的处理和 MatMul 层中的矩阵乘法似乎没有必要。

现在，我们创建一个从权重参数中抽取"单词 ID 对应行（向量）"的层，这里我们称之为 Embedding 层。顺便说一句，Embedding 来自"词嵌入"（word embedding）这一术语。也就是说，在这个 Embedding 层存放词嵌入（分布式表示）。



在自然语言处理领域，单词的密集向量表示称为词嵌入（word embedding）或者单词的分布式表示（distributed representation）。过去，将基于计数的方法获得的单词向量称为 distributional representation，将使用神经网络的基于推理的方法获得的单词向量称为 distributed representation。不过，中文里二者都译为"分布式表示"。

4.1.2 Embedding层的实现

从矩阵中取出某一行的处理是很容易实现的。这里，假设权重 W 是 NumPy 的二维数组。如果要从这个权重中取出某个特定的行，只需写 W[2] 或者 W[5]。用 Python 代码来实现，如下所示。

```
>>> import numpy as np
>>> W = np.arange(21).reshape(7, 3)
>>> W
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14],
       [15, 16, 17],
       [18, 19, 20]])
>>> W[2]
array([6, 7, 8])
```

```
>>> W[5]
array([15, 16, 17])
```

另外，从权重 W 中一次性提取多行的处理也很简单。只需通过数组指定行号即可，实际的代码如下所示。

```
>>> idx = np.array([1, 0, 3, 0])
>>> W[idx]
array([[ 3,  4,  5],
       [ 0,  1,  2],
       [ 9, 10, 11],
       [ 0,  1,  2]])
```

在这个例子中，我们一次性提取了4个索引（1、0、3、0）。通过将数组作为参数，可以一次性提取多行。顺便说一下，这里的实现假定用于mini-batch处理。

下面，我们来实现Embedding层的 forward() 方法。参照之前的例子，实现如下所示（✏ common/layers.py）。

```
class Embedding:
    def __init__(self, W):
        self.params = [W]
        self.grads = [np.zeros_like(W)]
        self.idx = None

    def forward(self, idx):
        W, = self.params
        self.idx = idx
        out = W[idx]
        return out
```

根据本书的代码规范，使用 params 和 grads 作为成员变量，并在成员变量 idx 中以数组的形式保存需要提取的行的索引（单词ID）。

接下来，我们考虑反向传播。Embedding层的正向传播只是从权重矩阵 W 中提取特定的行，并将该特定行的神经元原样传给下一层。因此，在反向传播时，从上一层（输出侧的层）传过来的梯度将原样传给下一层（输

入侧的层）。不过，从上一层传来的梯度会被应用到权重梯度 dW 的特定行（idx），如图 4-4 所示。

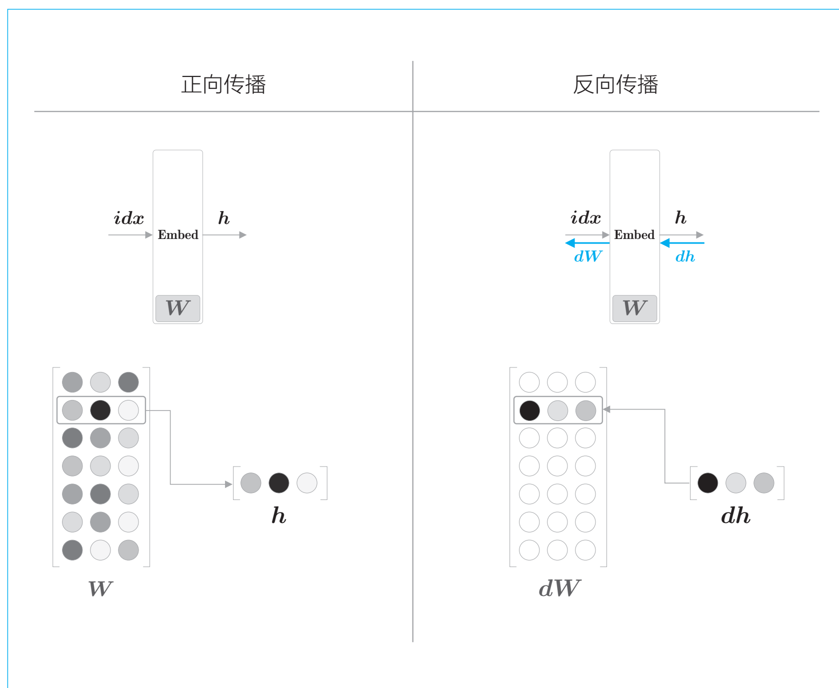


图 4-4　Embedding 层的正向传播和反向传播处理的概要（Embedding 层记为 Embed）

基于以上内容，我们来实现 backward()，代码如下所示。

```
def backward(self, dout):
    dW, = self.grads
    dW[...] = 0
    dW[self.idx] = dout # 不太好的方式
    return None
```

这里，取出权重梯度 dW，通过 dW[...] = 0 将 dW 的元素设为 0（并不是将 dW 设为 0，而是保持 dW 的形状不变，将它的元素设为 0）。然后，将上一层传来的梯度 dout 写入 idx 指定的行。



这里创建了和权重W相同大小的矩阵dW，并将梯度写入了dW对应的行。但是，我们最终想做的事情是更新权重W，所以没有必要特意创建dW(大小与W相同)。相反，只需把需要更新的行号(idx)及其对应的梯度(dout)保存下来，就可以更新权重(W)的特定行。但是，这里为了兼容已经实现的优化器类(Optimizer)，所以写成了现在的样子。

实际上，在刚才的 backward() 的实现中，存在一个问题，这一问题发生在 idx 的元素出现重复时。比如，当 idx 为 [0, 2, 0, 4] 时，就会发生图 4-5 中的问题。

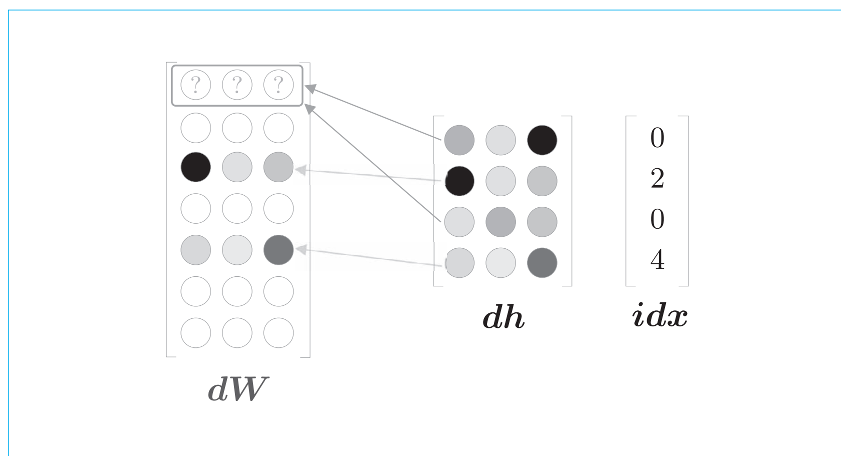


图4-5 当idx数组的元素中出现相同的行号时，简单地将dh的行写入对应位置就会有问题

如图 4-5 所示，我们将 dh 各行的值写入 dW 中 idx 指定的位置。在这种情况下，dW 的第 0 行会被写入两次。这样一来，其中某个值就会被覆盖掉。

为了解决这个重复问题，需要进行“加法”，而不是“写入”（请读者考虑一下为什么是加法）。也就是说，应该把 dh 各行的值累加到 dW 的对应行中。下面，我们来实现正确的反向传播。

```
def backward(self, dout):
    dW, = self.grads
```

```
dW[...] = 0

for i, word_id in enumerate(self.idx):
    dW[word_id] += dout[i]
# 或者
# np.add.at(dW, self.idx, dout)

return None
```

这里，我们使用 for 循环语句将梯度累加到对应索引上。这样一来，即便 idx 中出现了重复的索引，也能被正确处理。此外，这里使用 for 循环语句的实现也可以通过 NumPy 的 np.add.at() 进行。np.add.at(A, idx, B) 将 B 加到 A 上，此时可以通过 idx 指定 A 中需要进行加法的行。



通常情况下，NumPy 的内置方法比 Python 的 for 循环处理更快。这是因为 NumPy 的内置方法在底层做了高速化和提高处理效率的优化。因此，上面的代码如果使用 np.add.at() 来实现，效率会比使用 for 循环处理高得多。

关于 Embedding 层的实现就介绍到这里。现在，我们可以将 word2vec（CBOW 模型）的实现中的输入侧的 MatMul 层换成 Embedding 层。这样一来，既能减少内存使用量，又能避免不必要的计算。

4.2 word2vec 的改进②

下面，我们来进行 word2vec 的第 2 个改进。如前所述，word2vec 的另一个瓶颈在于中间层之后的处理，即矩阵乘积和 Softmax 层的计算。本节的目标就是解决这个瓶颈。这里，我们将采用名为负采样（negative sampling）的方法作为解决方案。使用 Negative Sampling 替代 Softmax，无论词汇量有多大，都可以使计算量保持较低或恒定。

本节的内容有些复杂，特别是实现方面会有点“纠结”。因此，我们会一个知识点一个知识点地确认，一步一步地前进。

4.2.1 中间层之后的计算问题

为了指出中间层之后的计算问题，和上一节一样，我们来考虑词汇量为100万个、中间层的神经元个数为100个的wod2vec（CBOW模型）。此时，word2vec进行的处理如图4-6所示。

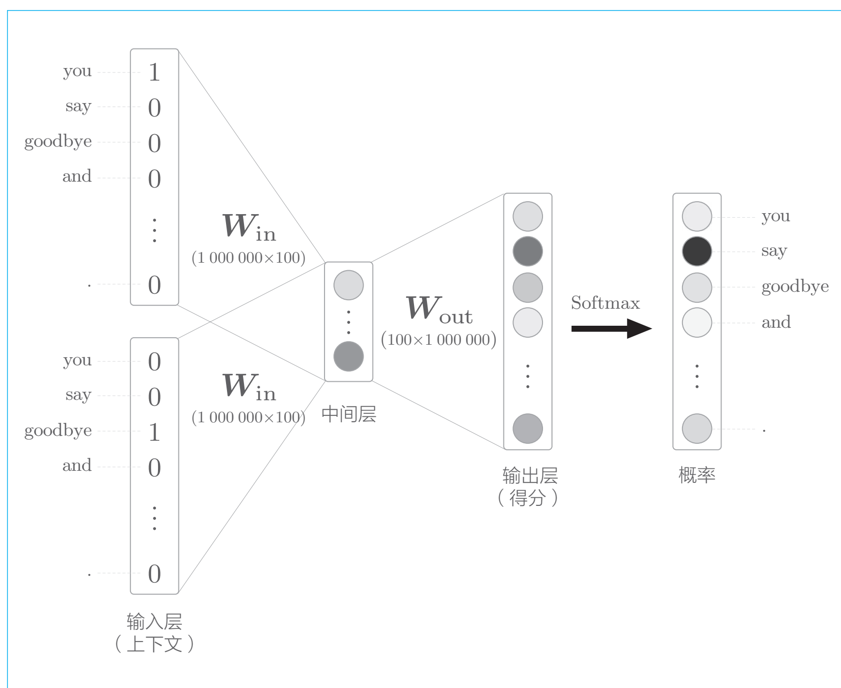


图4-6 词汇量为100万个时的word2vec：上下文是you和goodbye，目标词是say

如图4-6所示，输入层和输出层有100万个神经元。在上一节中，通过引入Embedding层，节省了输入层中不必要的计算。剩下的问题就是中间层之后的处理。此时，在以下两个地方需要很多计算时间。

- 中间层的神经元和权重矩阵（W_{out}）的乘积
- Softmax层的计算

第 1 个问题在于巨大的矩阵乘积计算。在上面的例子中，中间层向量的大小是 100，权重矩阵的大小是 100 × 1 000 000 万，如此巨大的矩阵乘积计算需要大量时间（也需要大量内存）。此外，因为反向传播时也要进行同样的计算，所以很有必要将矩阵乘积计算“轻量化”。

其次，Softmax 也会发生同样的问题。换句话说，随着词汇量的增加，Softmax 的计算量也会增加。观察 Softmax 的公式，就可以清楚地看出这一点。

$$y_k = \frac{\exp(s_k)}{\sum_{i=1}^{1\,000\,000} \exp(s_i)} \tag{4.1}$$

式 (4.1) 是第 k 个元素（单词）的 Softmax 的计算式（各个元素的得分为 $s_1, s_2, \cdots$）。因为假定词汇量是 100 万个，所以式 (4.1) 的分母需要进行 100 万次的 exp 计算。这个计算也与词汇量成正比，因此，需要一个可以替代 Softmax 的“轻量”的计算。

4.2.2　从多分类到二分类

下面，我们来解释一下负采样。这个方法的关键思想在于二分类（binary classification），更准确地说，是用二分类拟合多分类（multiclass classification），这是理解负采样的重点。

到目前为止，我们处理的都是多分类问题。拿刚才的例子来说，我们把它看作了从 100 万个单词中选择 1 个正确单词的任务。那么，可不可以将这个问题处理成二分类问题呢？更确切地说，我们是否可以用二分类问题来拟合这个多分类问题呢？



二分类处理的是答案为“Yes/No”的问题。诸如，“这个数字是 7 吗？”“这是猫吗？”“目标词是 say 吗？”等，这些问题都可以用“Yes/No”来回答。

到目前为止，我们已经做到了当给定上下文时，以较高的概率预测出作为正确解的单词。比如，当给定 you 和 goodbye 时，使神经网络预测出单词 say 的概率最高。如果学习进展顺利，神经网络就可以进行正确的预测。换句话说，对于“当上下文是 you 和 goodbye 时，目标词是什么？”这个问题，神经网络可以给出正确答案。

现在，我们来考虑如何将多分类问题转化为二分类问题。为此，我们先考察一个可以用“Yes/No”来回答的问题。比如，让神经网络来回答“当上下文是 you 和 goodbye 时，目标词是 say 吗？”这个问题，这时输出层只需要一个神经元即可。可以认为输出层的神经元输出的是 say 的得分。

那么，此时 CBOW 模型进行什么样的处理呢？用图来表示的话，可以画出图 4-7。

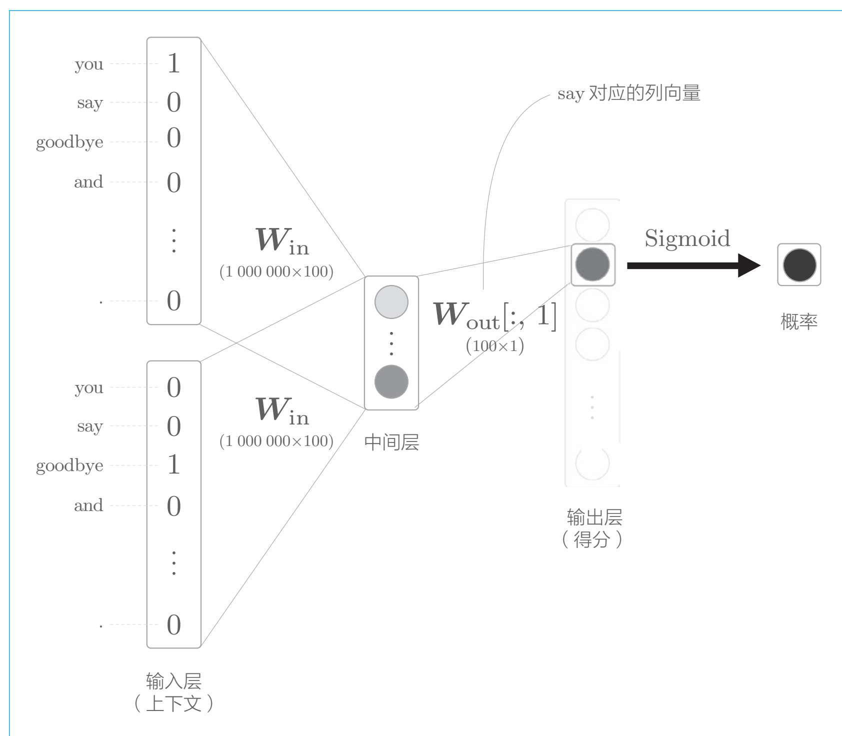


图 4-7　仅计算目标词的得分的神经网络

如图 4-7 所示，输出层的神经元仅有一个。因此，要计算中间层和输出侧的权重矩阵的乘积，只需要提取 say 对应的列（单词向量），并用它与中间层的神经元计算内积即可。这个计算的详细过程如图 4-8 所示。

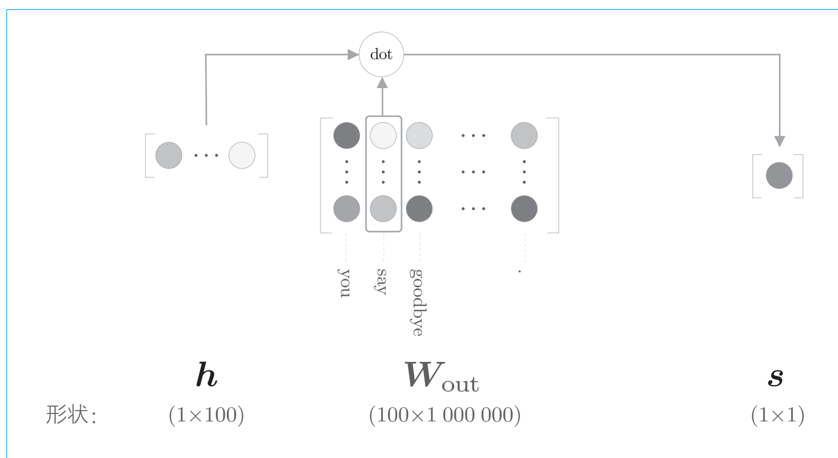


图 4-8　计算 say 对应的列向量和中间层的内积（图中的“dot”指内积运算）

如图 4-8 所示，输出侧的权重 $\boldsymbol{W}_{\text{out}}$ 中保存了各个单词 ID 对应的单词向量。此处，我们提取 say 这个单词向量，再求这个向量和中间层神经元的内积，这就是最终的得分。



到目前为止，输出层是以全部单词为对象进行计算的。这里，我们仅关注单词 say，计算它的得分。然后，使用 sigmoid 函数将其转化为概率。

4.2.3　sigmoid 函数和交叉熵误差

要使用神经网络解决二分类问题，需要使用 sigmoid 函数将得分转化为概率。为了求损失，我们使用交叉熵误差作为损失函数。这些都是二分类神经网络的老套路。



在多分类的情况下，输出层使用Softmax函数将得分转化为概率，损失函数使用交叉熵误差。在二分类的情况下，输出层使用sigmoid函数，损失函数也使用交叉熵误差。

这里我们先回顾一下 sigmoid 函数，如下式所示：

$$y = \frac{1}{1+\exp(-x)} \tag{4.2}$$

式 (4.2) 的图像如图 4-9 中的右图所示。从图中可以看出，sigmoid 函数呈 S 形，输入值 x 被转化为 0 到 1 之间的实数。这里的要点是，sigmoid 函数的输出 y 可以解释为概率。

另外，与 sigmoid 函数相关的 Sigmoid 层已经实现好了，如图 4-9 中的左图所示。

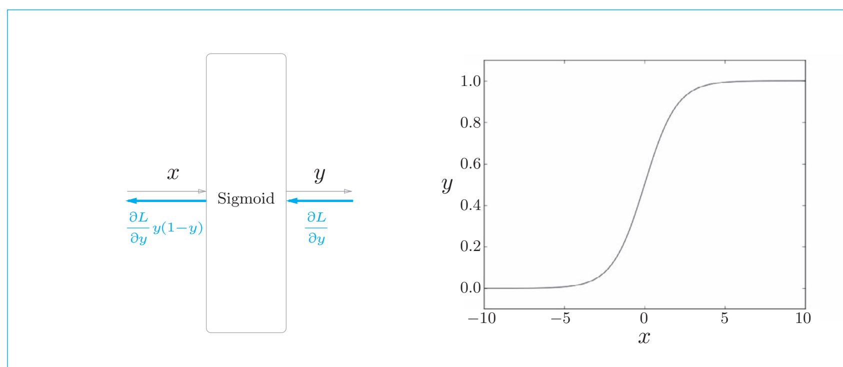


图4-9　Sigmoid层（左图）和sigmoid函数（右图）的图像

通过 sigmoid 函数得到概率 y 后，可以由概率 y 计算损失。与多分类一样，用于 sigmoid 函数的损失函数也是交叉熵误差，其数学式如下所示：

$$L = -(t \log y + (1-t) \log (1-y)) \tag{4.3}$$

其中，y 是 sigmoid 函数的输出，t 是正确解标签，取值为 0 或 1：取值为 1 时表示正确解是 “Yes”；取值为 0 时表示正确解是 “No”。因此，当 t 为 1 时，

输出 $-\log y$；当 t 为 0 时，输出 $-\log(1-y)$。



二分类和多分类的损失函数均为交叉熵误差，其数学式分别为式 (4.3) 和式 (1.7)。不过，它们只是写法不同而已，实际上表示的内容是一样的。确切地说，在多分类的情况下，如果输出层只有两个神经元，则式 (1.7) 和二分类的式 (4.3) 是完全一致的。因此，Sigmoid with Loss 层的实现只要在 Softmax with Loss 层的基础上稍加改动即可。

下面，我们用图来表示 Sigmoid 层和 Cross Entropy Error 层，如图 4-10 所示。

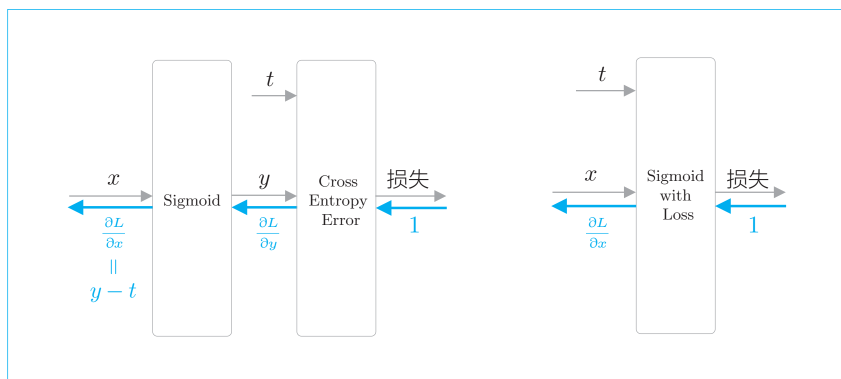


图 4-10　Sigmoid 层和 Cross Entropy Error 层的计算图。右图整合为了 Sigmoid with Loss 层

图 4-10 中值得注意的是反向传播的 $y-t$ 这个值。y 是神经网络输出的概率，t 是正确解标签，$y-t$ 正好是这两个值的差。这意味着，当正确解标签是 1 时，如果 y 尽可能地接近 1（100%），误差将很小。反过来，如果 y 远离 1，误差将增大。随后，这个误差向前面的层传播，当误差大时，模型学习得多；当误差小时，模型学习得少 [①]。

① 也就是说，误差越大，模型参数的更新力度就越大。——译者注



sigmoid函数和交叉熵误差的组合产生了$y-t$这样一个漂亮的结果。同样地，Softmax函数和交叉熵误差的组合，或者恒等函数和均方误差的组合也会在反向传播时传播$y-t$。

4.2.4 多分类到二分类的实现

下面，我们从实现的角度把之前讲的内容整理一下。前面我们处理了多分类问题，在输出层使用了与词汇量同等数量的神经元，并将它们传给了Softmax层。如果把重点放在“层”和“计算”上，则此时的神经网络可以画成图4-11。

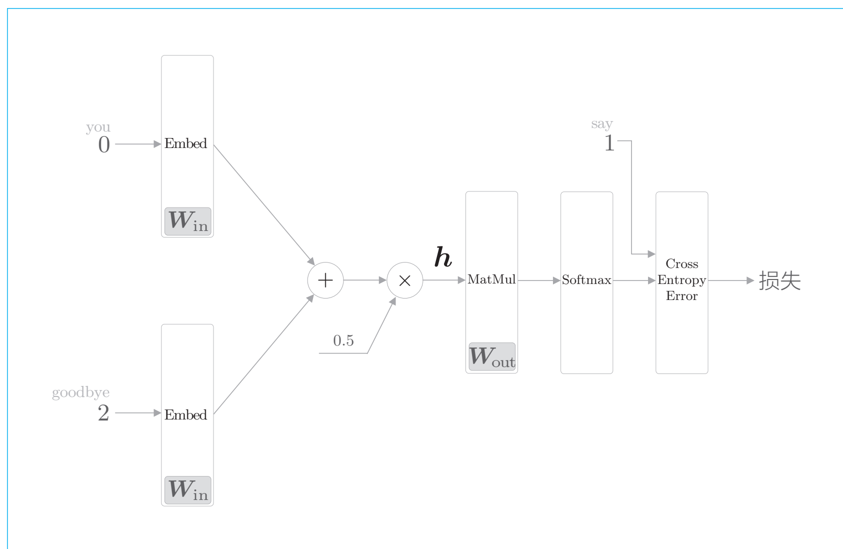


图4-11 进行多分类的CBOW模型的全貌图（Embedding层记为Embed）

图4-11中展示了上下文是you和goodbye、作为正确解的目标词是say的例子（假定you的单词ID是0，say的单词ID是1，goodbye的单词ID是2）。在输入层中，为了提取单词ID对应的分布式表示，使用了Embedding层。



上一节，我们实现了Embedding层，该层提取单词ID对应的分布式表示（单词向量）。以前我们用的是MatMul层。

现在，我们将图 4-11 中的神经网络转化成进行二分类的神经网络，网络结构如图 4-12 所示。

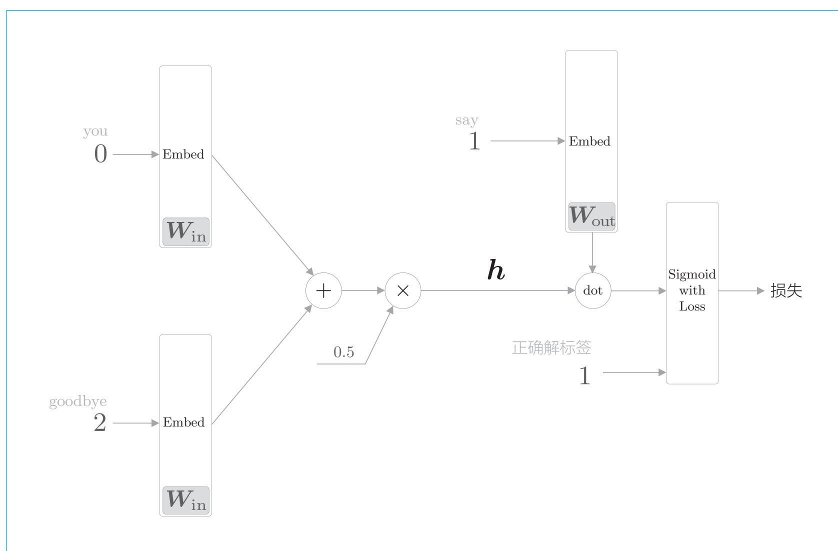


图 4-12　进行二分类的 CBOW 模型的全貌图

这里，将中间层的神经元记为 $\boldsymbol{h}$，并计算它与输出侧权重 $\boldsymbol{W}_{\text{out}}$ 中的单词 say 对应的单词向量的内积。然后，将其输出输入 Sigmoid with Loss 层，得到最终的损失。



在图 4-12 中，向Sigmoid with Loss层输入正确解标签1，这意味着现在正在处理的问题的答案是“Yes”。当答案是“No”时，向Sigmoid with Loss层输入0。

为了便于理解后面的内容，我们把图 4-12 的后半部分进一步简化。为此，我们引入 Embedding Dot 层，该层将图 4-12 中的 Embedding 层和 dot

运算（内积）合并起来处理。使用这个层，图 4-12 的后半部分可以画成图 4-13。

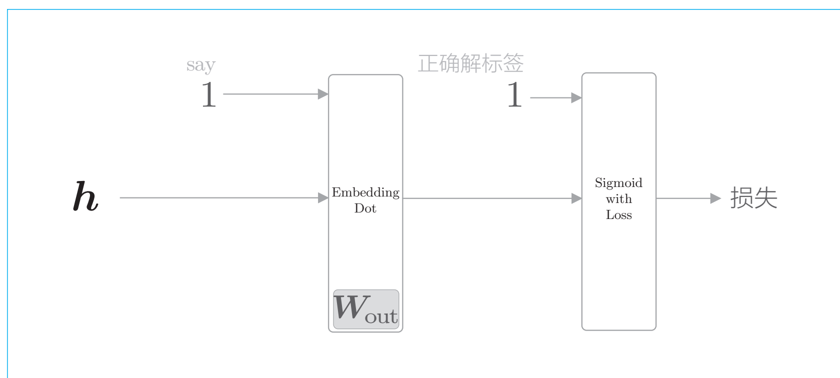


图 4-13　只关注图 4-12 的中间层之后的处理。使用 Embedding Dot 层合并 Embedding 层和内积运算

中间层的神经元 $\boldsymbol{h}$ 流经 Embedding Dot 层，传给 Sigmoid with Loss 层。从图中可以看出，使用 Embedding Dot 层之后，中间层之后的处理被简化了。

下面，我们简单地看一下 Embedding Dot 层的实现，这里我们将这个层实现为 EmbeddingDot 类（✏ ch04/negative_sampling_layer.py）。

```
class EmbeddingDot:
    def __init__(self, W):
        self.embed = Embedding(W)
        self.params = self.embed.params
        self.grads = self.embed.grads
        self.cache = None

    def forward(self, h, idx):
        target_W = self.embed.forward(idx)
        out = np.sum(target_W * h, axis=1)

        self.cache = (h, target_W)
        return out
```

```python
    def backward(self, dout):
        h, target_W = self.cache
        dout = dout.reshape(dout.shape[0], 1)

        dtarget_W = dout * h
        self.embed.backward(dtarget_W)
        dh = dout * target_W
        return dh
```

EmbeddingDot 类共有 4 个成员变量：embed、params、grads 和 cache。根据本书的代码规范，params 保存参数，grads 保存梯度。另外，作为缓存，embed 保存 Embedding 层，cache 保存正向传播时的计算结果。

正向传播的 forward(h, idx) 方法的参数接收中间层的神经元（h）和单词 ID 的 NumPy 数组（idx）。这里，idx 是单词 ID 列表，这是因为我们假定了对数据进行 mini-batch 处理。

在上面的代码中，forward() 方法首先调用 Embedding 层的 forward(idx) 方法，然后通过 np.sum(self. target_W * h, axis=1) 计算内积。通过观察具体值来理解这个实现会比较快，如图 4-14 所示。

```
embed = Embedding(W)
target_W = embed.forward(idx)
out = np.sum(target_W * h, axis=1)
```

W	idx	target_W	h	target_W * h	out
[[0 1 2]	[0 3 1]	[[0 1 2]	[[0 1 2]	[[0 1 4]	[5 122 86]
[3 4 5]		[9 10 11]	[3 4 5]	[27 40 55]	
[6 7 8]		[3 4 5]]	[6 7 8]]	[18 28 40]]	
[9 10 11]					
[12 13 14]					
[15 16 17]					
[18 19 20]]					

图 4-14　Embedding Dot 层中各个变量的值

如图 4-14 所示，准备适当的 W、h 和 idx。这里，idx 是 [0, 3, 1]，这个例子表示 mini-batch 一并处理 3 笔数据。因为 idx 是 [0, 3, 1]，所以 target_W 将提取出 W 的第 0 行、第 3 行和第 1 行。另外，target_W * h 计算

对应元素的乘积（NumPy 的 “*” 计算对应元素的乘积）。然后，对结果逐行（`axis=1`）进行求和，得到最终的结果 `out`。

以上就是对 Embedding Dot 层的正向传播的介绍。反向传播以相反的顺序传播梯度，这里我们省略对其实现的说明（并不是特别难，请大家自己思考）。

4.2.5 负采样

至此，我们成功地把要解决的问题从多分类问题转化成了二分类问题。但是，这样问题就被解决了吗？很遗憾，事实并非如此。因为我们目前仅学习了正例（正确答案），还不确定负例（错误答案）会有怎样的结果。

现在，我们再来思考一下之前的例子。在之前的例子中，上下文是 you 和 goodbye，目标词是 say。我们到目前为止只是对正例 say 进行了二分类，如果此时模型有 “好的权重”，则 Sigmoid 层的输出（概率）将接近 1。用计算图来表示此时的处理，如图 4-15 所示。

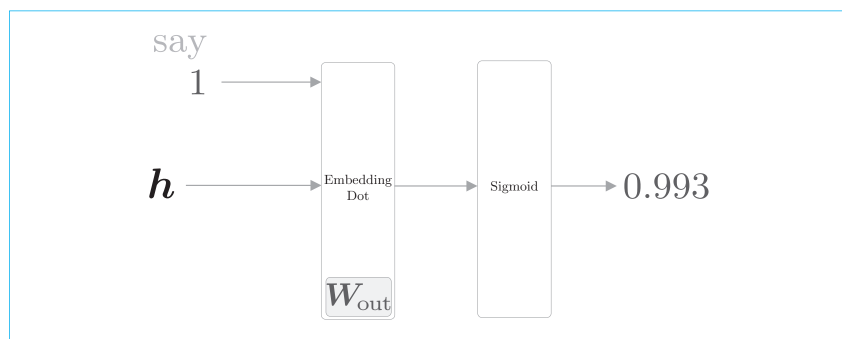


图 4-15 CBOW 模型的中间层之后的处理：上下文是 you 和 goodbye，此时目标词是 say 的概率为 0.993（99.3%）

当前的神经网络只是学习了正例 say，但是对 say 之外的负例一无所知。而我们真正要做的事情是，对于正例（say），使 Sigmoid 层的输出接近 1；对于负例（say 以外的单词），使 Sigmoid 层的输出接近 0。用图来表示，如图 4-16 所示。

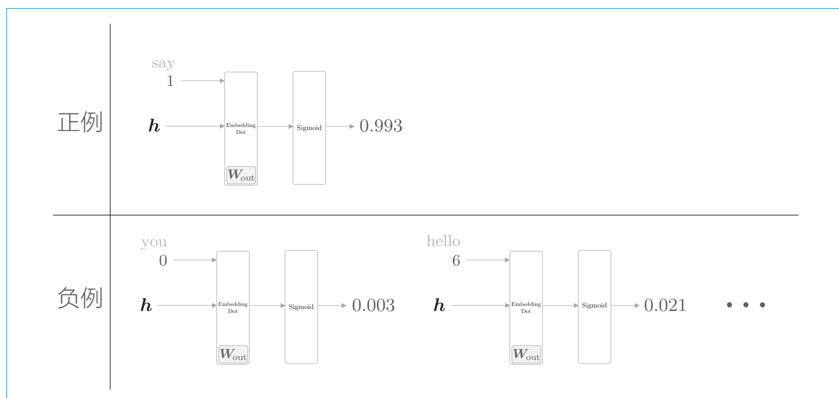


图4-16　如果say是正例（正确答案），则当输入say时使Sigmoid层的输出接近1，当输入say以外的单词时使输出接近0，我们要求的是这样的权重

比如，当上下文是you和goodbye时，我们希望目标词是hello（错误答案）的概率较低。在图4-16中，目标词是hello的概率为0.021（2.1%），我们要求的就是这种能使输出接近0的权重。



为了把多分类问题处理为二分类问题，对于“正确答案”（正例）和“错误答案”（负例），都需要能够正确地进行分类（二分类）。因此，需要同时考虑正例和负例。

那么，我们需要以所有的负例为对象进行学习吗？答案显然是“No”。如果以所有的负例为对象，词汇量将暴增至无法处理（更何况本章的目的本来就是解决词汇量增加的问题）。为此，作为一种近似方法，我们将选择若干个（5个或者10个）负例（如何选择将在下文介绍）。也就是说，只使用少数负例。这就是负采样方法的含义。

总而言之，负采样方法既可以求将正例作为目标词时的损失，同时也可以采样（选出）若干个负例，对这些负例求损失。然后，将这些数据（正例和采样出来的负例）的损失加起来，将其结果作为最终的损失。

下面，让我们结合具体的例子来说明。这里使用与之前相同的例子（正例目标词是say）。假设选取2个负例目标词hello和i，此时，如果我们只

关注 CBOW 模型的中间层之后的部分，则负采样的计算图如图 4-17 所示。

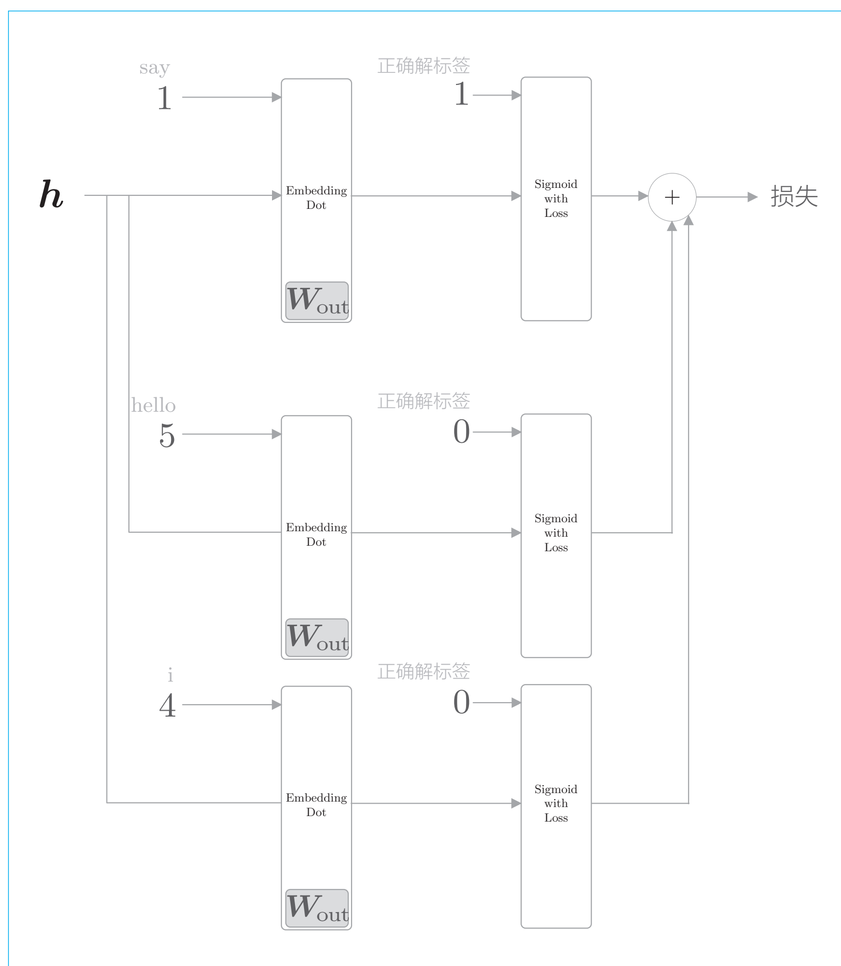


图 4-17 负采样的例子（只关注中间层之后的处理，画出基于层的计算图）

图 4-17 中需要注意的是对正例和负例的处理。正例（say）和之前一样，向 Sigmoid with Loss 层输入正确解标签 1；而因为负例（hello 和 i）是错误答案，所以要向 Sigmoid with Loss 层输入正确解标签 0。此后，将各个数据的损失相加，作为最终损失输出。

4.2.6 负采样的采样方法

下面我们来看一下如何抽取负例。关于这一点，基于语料库的统计数据进行采样的方法比随机抽样要好。具体来说，就是让语料库中经常出现的单词容易被抽到，让语料库中不经常出现的单词难以被抽到。

基于语料库中单词使用频率的采样方法会先计算语料库中各个单词的出现次数，并将其表示为“概率分布”，然后使用这个概率分布对单词进行采样（图 4-18）。

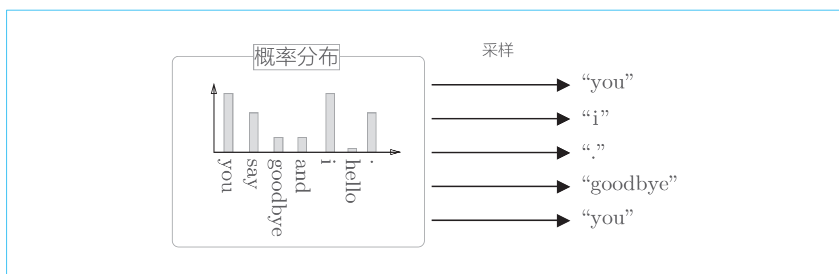


图 4-18　根据概率分布多次进行采样的例子

基于语料库中各个单词的出现次数求出概率分布后，只需根据这个概率分布进行采样就可以了。通过根据概率分布进行采样，语料库中经常出现的单词将容易被抽到，而“稀有单词”将难以被抽到。



负采样应当尽可能多地覆盖负例单词，但是考虑到计算的复杂度，有必要将负例限定在较小范围内（5 个或者 10 个）。这里，如果只选择稀有单词作为负例会怎样呢？结果会很糟糕。因为在现实问题中，稀有单词基本上不会出现。也就是说，处理稀有单词的重要性较低。相反，处理好高频单词才能获得更好的结果。

下面，我们使用 Python 来说明基于概率分布的采样。为此，可以使用 NumPy 的 `np.random.choice()` 方法。这里我们结合几个具体的例子来看一下这个方法的用法。

```
>>> import numpy as np

# 从0到9的数字中随机选择一个数字
>>> np.random.choice(10)
7
>>> np.random.choice(10)
2

# 从words列表中随机选择一个元素
>>> words = ['you', 'say', 'goodbye', 'I', 'hello', '.']
>>> np.random.choice(words)
'goodbye'

# 有放回采样5次
>>> np.random.choice(words, size=5)
array(['goodbye', '.', 'hello', 'goodbye', 'say'],
      dtype='<U7')

# 无放回采样5次
>>> np.random.choice(words, size=5, replace=False)
array(['hello', '.', 'goodbye', 'I', 'you'],
      dtype='<U7')

# 基于概率分布进行采样
>>> p = [0.5, 0.1, 0.05, 0.2, 0.05, 0.1]
>>> np.random.choice(words, p=p)
'you'
```

如上所示，`np.random.choice()` 可以用于随机抽样。如果指定 `size` 参数，将执行多次采样。如果指定 `replace=False`，将进行无放回采样。通过给参数 `p` 指定表示概率分布的列表，将进行基于概率分布的采样。剩下的就是使用这个函数抽取负例。

word2vec 中提出的负采样对刚才的概率分布增加了一个步骤。如式 (4.4) 所示，对原来的概率分布取 0.75 次方。

$$P'(w_i) = \frac{P(w_i)^{0.75}}{\sum\limits_j^n P(w_j)^{0.75}} \tag{4.4}$$

这里，$P(w_i)$ 表示第 i 个单词的概率。式 (4.4) 只是对原来的概率分布的各个元素取 0.75 次方。不过，为了使变换后的概率总和仍为 1，分母需要变成“变换后的概率分布的总和”。

那么，为什么我们要进行式 (4.4) 的变换呢？这是为了防止低频单词被忽略。更准确地说，通过取 0.75 次方，低频单词的概率将稍微变高。我们来看一个具体例子，如下所示。

```
>>> p = [0.7, 0.29, 0.01]
>>> new_p = np.power(p, 0.75)
>>> new_p /= np.sum(new_p)
>>> print(new_p)
[ 0.64196878  0.33150408  0.02652714]
```

根据这个例子，变换前概率为 0.01（1%）的元素，变换后为 0.026...（2.6...%）。通过这种方式，取 0.75 次方作为一种补救措施，使得低频单词稍微更容易被抽到。此外，0.75 这个值并没有什么理论依据，也可以设置成 0.75 以外的值。

如上所示，负采样从语料库生成单词的概率分布，在取其 0.75 次方之后，再使用之前的 np.random.choice() 对负例进行采样。本书中将这些处理实现为了 UnigramSampler 类。这里仅简单说明 UnigramSampler 类的使用方法，其具体实现在 ch04/negative_sampling_layer.py 中，对细节感兴趣的读者可以参考一下。



unigram 是“1 个(连续)单词”的意思。同样地，bigram 是“2 个连续单词”的意思，trigram 是“3 个连续单词”的意思。这里使用 UnigramSampler 这个名字，是因为我们以 1 个单词为对象创建概率分布。如果是 bigram，则以 (‘you’, ‘say’)、(‘you’, ‘goodbye’)……这样的 2 个单词的组合为对象创建概率分布。

在进行初始化时，UnigramSampler 类取 3 个参数，分别是单词 ID 列表格式的 corpus、对概率分布取的次方值 power（默认值是 0.75）和负例的采样个数 sample_size。UnigramSampler 类有 get_negative_sample(target) 方法，该方法以参数 target 指定的单词 ID 为正例，对其他的单词 ID 进行采样。下面，我们来看一个简单的 UnigramSampler 类的使用示例。

```
corpus = np.array([0, 1, 2, 3, 4, 1, 2, 3])
power = 0.75
sample_size = 2

sampler = UnigramSampler(corpus, power, sample_size)
target = np.array([1, 3, 0])
negative_sample = sampler.get_negative_sample(target)
print(negative_sample)
# [[0 3]
#  [1 2]
#  [2 3]]
```

这里，考虑将 [1, 3, 0] 这 3 个数据的 mini-batch 作为正例。此时，对各个数据采样 2 个负例。在上面的例子中，可知第 1 个数据的负例是 [0, 3]，第 2 个是 [1, 2]，第 3 个是 [2, 3]。这样一来，我们就完成了负采样。

4.2.7 负采样的实现

最后，我们来实现负采样。我们把它实现为 NegativeSamplingLoss 类，首先从初始化开始（✏ ch04/negative_sampling_layer.py）。

```
class NegativeSamplingLoss:
    def __init__(self, W, corpus, power=0.75, sample_size=5):
        self.sample_size = sample_size
        self.sampler = UnigramSampler(corpus, power, sample_size)
        self.loss_layers = [SigmoidWithLoss() for _ in range(sample_size
↪   + 1)]
        self.embed_dot_layers = [EmbeddingDot(W) for _ in
↪   range(sample_size + 1)]
```

```
        self.params, self.grads = [], []
        for layer in self.embed_dot_layers:
            self.params += layer.params
            self.grads += layer.grads
```

初始化的参数有表示输出侧权重的 W、语料库（单词 ID 列表）corpus、概率分布的次方值 power 和负例的采样数 sample_size。这里，生成上一节所说的 UnigramSampler 类，并使用成员变量 sampler 保存。另外，将负例的采样数设置为成员变量 sample_size。

成员变量 loss_layers 和 embed_dot_layers 中以列表格式保存了必要的层。在这两个列表中生成 sample_size + 1 个层，这是因为需要生成一个正例用的层和 sample_size 个负例用的层。这里，我们假设列表的第一个层处理正例。也就是说，loss_layers[0] 和 embed_dot_layers[0] 是处理正例的层。然后，将 embed_dot_layers 层使用的权重和梯度分别保存在数组中。下面，我们给出正向传播的实现（✏ ch04/negative_sampling_layer.py）。

```
    def forward(self, h, target):
        batch_size = target.shape[0]
        negative_sample = self.sampler.get_negative_sample(target)

        # 正例的正向传播
        score = self.embed_dot_layers[0].forward(h, target)
        correct_label = np.ones(batch_size, dtype=np.int32)
        loss = self.loss_layers[0].forward(score, correct_label)

        # 负例的正向传播
        negative_label = np.zeros(batch_size, dtype=np.int32)
        for i in range(self.sample_size):
            negative_target = negative_sample[:, i]
            score = self.embed_dot_layers[1 + i].forward(h, negative_target)
            loss += self.loss_layers[1 + i].forward(score, negative_label)

        return loss
```

forward(h, target) 方法接收的参数是中间层的神经元 h 和正例目标词 target。这里进行的处理是，首先使用 self.sampler 采样负例，并设为

negative_sample。然后，分别对正例和负例的数据进行正向传播，求损失的和。具体而言，通过 Embedding Dot 层的 forward 输出得分，再将这个得分和标签一起输入 Sigmoid with Loss 层来计算损失。这里需要注意的是，正例的正确解标签为 1，负例的正确解标签为 0。

最后，我们来看反向传播的实现。

```
def backward(self, dout=1):
    dh = 0
    for l0, l1 in zip(self.loss_layers, self.embed_dot_layers):
        dscore = l0.backward(dout)
        dh += l1.backward(dscore)
    return dh
```

反向传播的实现非常简单，只需要以与正向传播相反的顺序调用各层的 backward() 函数即可。在正向传播时，中间层的神经元被复制了多份，这相当于 1.3.4.3 节中介绍的 Repeat 节点。因此，在反向传播时，需要将多份梯度累加起来。以上就是负采样的实现的说明。

4.3　改进版 word2vec 的学习

到目前为止，我们进行了 word2vec 的改进。首先说明了 Embedding 层，又介绍了负采样的方法，然后对这两者进行了实现。现在我们进一步来实现进行了这些改进的神经网络，并在 PTB 数据集上进行学习，以获得更加实用的单词的分布式表示。

4.3.1　CBOW 模型的实现

这里，我们将改进上一章的简单的 SimpleCBOW 类，来实现 CBOW 模型。改进之处在于使用 Embedding 层和 Negative Sampling Loss 层。此外，我们将上下文部分扩展为可以处理任意的窗口大小。

改进版的 CBOW 类的实现如下所示。首先，我们来看一下初始化方法（✏ ch04/cbow.py）。

```
import sys
sys.path.append('..')
import numpy as np
from common.layers import Embedding
from ch04.negative_sampling_layer import NegativeSamplingLoss


class CBOW:
    def __init__(self, vocab_size, hidden_size, window_size, corpus):
        V, H = vocab_size, hidden_size

        # 初始化权重
        W_in = 0.01 * np.random.randn(V, H).astype('f')
        W_out = 0.01 * np.random.randn(V, H).astype('f')

        # 生成层
        self.in_layers = []
        for i in range(2 * window_size):
            layer = Embedding(W_in)  # 使用Embedding层
            self.in_layers.append(layer)
        self.ns_loss = NegativeSamplingLoss(W_out, corpus, power=0.75,
↪   sample_size=5)

        # 将所有的权重和梯度整理到列表中
        layers = self.in_layers + [self.ns_loss]
        self.params, self.grads = [], []
        for layer in layers:
            self.params += layer.params
            self.grads += layer.grads

        # 将单词的分布式表示设置为成员变量
        self.word_vecs = W_in
```

这个初始化方法有 4 个参数。vocab_size 是词汇量，hidden_size 是中间层的神经元个数，corpus 是单词 ID 列表。另外，通过 window_size 指定上下文的大小，即上下文包含多少个周围单词。如果 window_size 是 2，则目标词的左右 2 个单词（共 4 个单词）将成为上下文。



在SimpleCBOW类(改进前的实现)中，输入侧的权重和输出侧的权重的形状不同，输出侧的权重在列方向上排列单词向量。而CBOW类的输出侧的权重和输入侧的权重形状相同，都在行方向上排列单词向量。这是因为NegativeSamplingLoss类中使用了Embedding层。

在权重的初始化结束后，继续创建层。这里，创建 2 * window_size 个 Embedding 层，并将其保存在成员变量 in_layers 中。然后，创建 Negative Sampling Loss 层。

在创建好层之后，将神经网络中使用的参数和梯度放入成员变量 params 和 grads 中。另外，为了之后可以访问单词的分布式表示，将权重 W_in 设置为成员变量 word_vecs。下面，我们来看一下正向传播的 forward() 方法和反向传播的 backward() 方法（✏ ch04/cbow.py）。

```
def forward(self, contexts, target):
    h = 0
    for i, layer in enumerate(self.in_layers):
        h += layer.forward(contexts[:, i])
    h *= 1 / len(self.in_layers)
    loss = self.ns_loss.forward(h, target)
    return loss

def backward(self, dout=1):
    dout = self.ns_loss.backward(dout)
    dout *= 1 / len(self.in_layers)
    for layer in self.in_layers:
        layer.backward(dout)
    return None
```

这里的实现只是按适当的顺序调用各个层的正向传播（或反向传播），这是对上一章的 SimpleCBOW 类的自然扩展。不过，虽然 forward (contexts, target) 方法取的参数仍是上下文和目标词，但是它们是单词 ID 形式的（上一章中使用的是 one-hot 向量，不是单词 ID），具体示例如图 4-19 所示。

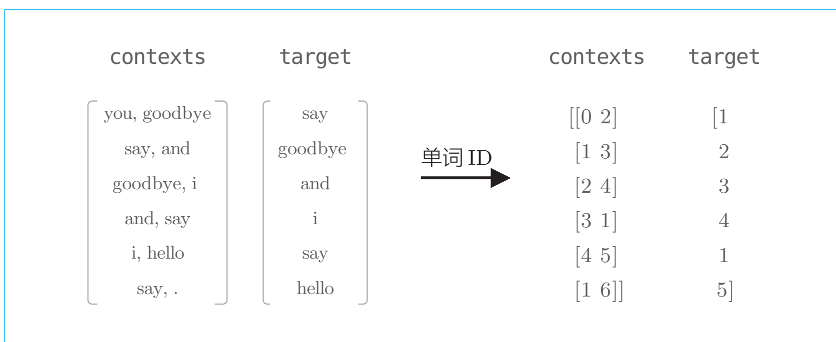


图4-19　用单词ID表示上下文和目标词的例子：这里显示的是窗口大小为1的上下文

图4-19的右侧显示的单词ID列表是`contexts`和`target`的例子。可以看出，`contexts`是一个二维数组，`target`是一个一维数组，这样的数据被输入`forward(contexts, target)`中。以上就是`CBOW`类的说明。

4.3.2　CBOW模型的学习代码

最后，我们来实现CBOW模型的学习部分。其实只是复用一下神经网络的学习，如下所示（✏ `ch04/train.py`）。

```
import sys
sys.path.append('..')
import numpy as np
from common import config
# 在用GPU运行时，请打开下面的注释（需要cupy）
# ===============================================
# config.GPU = True
# ===============================================
import pickle
from common.trainer import Trainer
from common.optimizer import Adam
from cbow import CBOW
from common.util import create_contexts_target, to_cpu, to_gpu
from dataset import ptb

# 设定超参数
```

```
window_size = 5
hidden_size = 100
batch_size = 100
max_epoch = 10

# 读入数据
corpus, word_to_id, id_to_word = ptb.load_data('train')
vocab_size = len(word_to_id)

contexts, target = create_contexts_target(corpus, window_size)
if config.GPU:
    contexts, target = to_gpu(contexts), to_gpu(target)

# 生成模型等
model = CBOW(vocab_size, hidden_size, window_size, corpus)
optimizer = Adam()
trainer = Trainer(model, optimizer)

# 开始学习
trainer.fit(contexts, target, max_epoch, batch_size)
trainer.plot()

# 保存必要数据，以便后续使用
word_vecs = model.word_vecs
if config.GPU:
    word_vecs = to_cpu(word_vecs)
params = {}
params['word_vecs'] = word_vecs.astype(np.float16)
params['word_to_id'] = word_to_id
params['id_to_word'] = id_to_word
pkl_file = 'cbow_params.pkl'
with open(pkl_file, 'wb') as f:
    pickle.dump(params, f, -1)
```

本次的 CBOW 模型的窗口大小为 5，隐藏层的神经元个数为 100。虽然具体取决于语料库的情况，但是一般而言，当窗口大小为 2 ~ 10、中间层的神经元个数（单词的分布式表示的维数）为 50 ~ 500 时，结果会比较好。稍后我们会对这些超参数进行讨论。

这次我们利用的 PTB 语料库比之前要大得多，因此学习需要很长时间（半天左右）。作为一种选择，我们提供了使用 GPU 运行的模式。如果要使用 GPU 运行，需要打开顶部的“`# config.GPU = True`”。不过，使用 GPU 运行需要有一台安装了 NVIDIA GPU 和 CuPy 的机器。

在学习结束后，取出权重（输入侧的权重），并保存在文件中以备后用（用于单词和单词 ID 之间的转化的字典也一起保存）。这里，使用 Python 的 pickle 功能进行文件保存。pickle 可以将 Python 代码中的对象保存到文件中（或者从文件中读取对象）。



ch04/cbow_params.pkl 中提供了学习好的参数。如果不想等学习结束，可以使用本书提供的学习好的参数。根据学习环境的不同，学习到的权重数据也不一样。这是由权重初始化时用到的随机初始值、mini-bath 的随机选取，以及负采样的随机抽样造成的。因为这些随机性，最后得到的权重在各自的环境中会不一样。不过宏观来看，得到的结果(趋势)是类似的。

4.3.3 CBOW 模型的评价

现在，我们来评价一下上一节学习到的单词的分布式表示。这里我们使用第 2 章中实现的 `most_similar()` 函数，显示几个单词的最接近的单词（✏ ch04/eval.py）。

```
import sys
sys.path.append('..')
from common.util import most_similar
import pickle


pkl_file = 'cbow_params.pkl'

with open(pkl_file, 'rb') as f:
    params = pickle.load(f)
    word_vecs = params['word_vecs']
    word_to_id = params['word_to_id']
```

```
    id_to_word = params['id_to_word']

querys = ['you', 'year', 'car', 'toyota']
for query in querys:
    most_similar(query, word_to_id, id_to_word, word_vecs, top=5)
```

运行上面的代码，可以得以下结果（具体结果会根据各自的学习环境而有所差异）。

```
[query] you
 we: 0.610597074032
 someone: 0.591710150242
 i: 0.554366409779
 something: 0.490028560162
 anyone: 0.473472118378

[query] year
 month: 0.718261063099
 week: 0.652263045311
 spring: 0.62699586153
 summer: 0.625829637051
 decade: 0.603022158146

[query] car
 luxury: 0.497202396393
 arabia: 0.478033810854
 auto: 0.471043765545
 disk-drive: 0.450782179832
 travel: 0.40902107954

[query] toyota
 ford: 0.550541639328
 instrumentation: 0.510020911694
 mazda: 0.49361255765
 bethlehem: 0.474817842245
 nissan: 0.474622786045
```

我们看一下结果。首先，在查询 you 的情况下，近似单词中出现了人称代词 i（＝I）和 we 等。接着，查询 year，可以看到 month、week 等表

示时间区间的具有相同性质的单词。然后，查询 toyota，可以得到 ford、mazda 和 nissan 等表示汽车制造商的词汇。从这些结果可以看出，由 CBOW 模型获得的单词的分布式表示具有良好的性质。

此外，由 word2vec 获得的单词的分布式表示不仅可以将近似单词聚拢在一起，还可以捕获更复杂的模式，其中一个具有代表性的例子是因“king − man + woman = queen”而出名的类推问题（类比问题）。更准确地说，使用 word2vec 的单词的分布式表示，可以通过向量的加减法来解决类推问题。

如图 4-20 所示，要解决类推问题，需要在单词向量空间上寻找尽可能使“man → woman”向量和“king → ?”向量接近的单词。

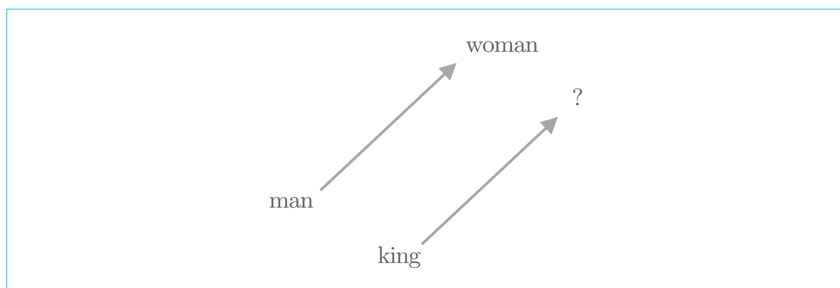


图 4-20 借助“man : woman = king : ?”这个类推问题，展示各个单词在单词向量空间上的相关性

这里用 vec(‘man’) 表示单词 man 的分布式表示（单词向量）。如此一来，图 4-20 中要求的关联性可以用数学式表示为 vec(‘woman’) − vec(‘man’) = vec(?) − vec(‘king’)。将其变形，有 vec(‘king’) + vec(‘woman’) − vec(‘man’) = vec(?)。也就是说，我们的任务是找到离向量 vec(‘king’) + vec(‘woman’) − vec(‘man’) 最近的单词向量。本书在 `common/util.py` 中提供了实现此逻辑的函数 `analogy()`。使用这个函数，可以用 `analogy('man', 'king', 'woman', word_to_id, id_to_word, word_vecs, top=5)` 这样 1 行代码来回答刚才的类推问题。此时，输出的结果如下所示。

```
[analogy] man:king = woman:?
 word1: 5.003233
 word2: 4.400302
 word3: 4.22342
 word4: 4.003234
 word5: 3.934550
```

第 1 行显示的是问题，之后按照得分从高到低的顺序输出 5 个单词，得分显示在各个单词的旁边。现在，我们来实际解决几个类推问题。下面是 4 个问题（✏ ch04/eval.py）。

```
analogy('king', 'man', 'queen',  word_to_id, id_to_word, word_vecs)
analogy('take', 'took', 'go',  word_to_id, id_to_word, word_vecs)
analogy('car', 'cars', 'child',  word_to_id, id_to_word, word_vecs)
analogy('good', 'better', 'bad',  word_to_id, id_to_word, word_vecs)
```

执行这些代码，可以得到如下结果。

```
[analogy] king:man = queen:?
 woman: 5.161407947540283
 veto: 4.928170680999756
 ounce: 4.689689636230469
 earthquake: 4.633471488952637
 successor: 4.6089653968811035

[analogy] take:took = go:?
 went: 4.548568248748779
 points: 4.248863220214844
 began: 4.090967178344727
 comes: 3.9805688858032227
 oct.: 3.9044761657714844

[analogy] car:cars = child:?
 children: 5.217921257019043
 average: 4.725458145141602
 yield: 4.208011627197266
 cattle: 4.18687629699707
 priced: 4.178797245025635
```

```
[analogy] good:better = bad:?
 more: 6.647829532623291
 less: 6.063825607299805
 rather: 5.220577716827393
 slower: 4.733833312988281
 greater: 4.672840118408203
```

结果符合我们的预期。第 1 个问题是 “king : man = queen : ?”，这里正确地回答了 “woman”。第 2 个问题是 “take : took = go : ?”，也按预期回答了 “went”。这是捕获了现在时和过去时之间的模式的证据，可以解释为单词的分布式表示编码了时态相关的信息。从第 3 题可知，单词的单数形式和复数形式之间的模式也被正确地捕获。可惜的是，对于第 4 题 “good : better = bad : ?”，并没能回答出 “worse”。不过，看到 more、less 等比较级的单词出现在回答中，说明这些性质也被编码在了单词的分布式表示中。

像这样，使用 word2vec 获得的单词的分布式表示，可以通过向量的加减法求解类推问题。不仅限于单词的含义，它也捕获了语法中的模式。另外，我们还在 word2vec 的单词的分布式表示中发现了一些有趣的结果，比如 good 和 best 之间存在 better 这样的关系。



这里的类推问题的结果看上去非常好。不过遗憾的是，这是笔者特意选出来的能够被顺利解决的问题。实际上，很多问题都无法获得预期的结果。这是因为 PTB 数据集的规模还是比较小。如果使用更大规模的语料库，可以获得更准确、更可靠的单词的分布式表示，从而大大提高类推问题的准确率。

4.4　wor2vec相关的其他话题

关于 word2vec 的机制和实现，我们差不多都介绍完了。本节我们来讨论一些有关 word2cev 的其他话题。

4.4.1 word2vec的应用例

使用 word2vec 获得的单词的分布式表示可以用来查找近似单词，但是单词的分布式表示的好处不仅仅在于此。在自然语言处理领域，单词的分布式表示之所以重要，原因就在于迁移学习（transfer learning）。迁移学习是指在某个领域学到的知识可以被应用于其他领域。

在解决自然语言处理任务时，一般不会使用 word2vec 从零开始学习单词的分布式表示，而是先在大规模语料库（Wikipedia、Google News 等文本数据）上学习，然后将学习好的分布式表示应用于某个单独的任务。比如，在文本分类、文本聚类、词性标注和情感分析等自然语言处理任务中，第一步的单词向量化工作就可以使用学习好的单词的分布式表示。在几乎所有类型的自然语言处理任务中，单词的分布式表示都有很好的效果！

单词的分布式表示的优点是可以将单词转化为固定长度的向量。另外，使用单词的分布式表示，也可以将文档（单词序列）转化为固定长度的向量。目前，关于如何将文档转化为固定长度的向量，相关研究已经进行了很多，最简单的方法是，把文档的各个单词转化为分布式表示，然后求它们的总和。这是一种被称为 bag-of-words 的不考虑单词顺序的模型（思想）。此外，使用即将在第 5 章中说明的循环神经网络，可以以更加优美的方式利用 word2vec 的单词的分布式表示来将文档转化为固定长度的向量。

将单词和文档转化为固定长度的向量是非常重要的。因为如果可以将自然语言转化为向量，就可以使用常规的机器学习方法（神经网络、SVM 等），如图 4-21 所示。

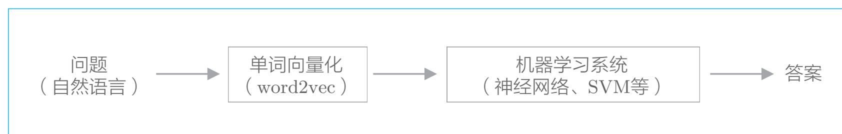


图 4-21 使用了单词的分布式表示的系统的处理流程

由图 4-21 可知，如果可以将自然语言书写的问题转化为固定长度的向

量，就可以将这个向量作为其他机器学习系统的输入。通过将自然语言转化为向量，可以利用常规的机器学习框架输出目标答案（包括它的学习）。



在图 4-21 的流程中，单词的分布式表示的学习和机器学习系统的学习通常使用不同的数据集独立进行。比如，单词的分布式表示使用 Wikipedia 等通用语料库预先学习好，然后机器学习系统（SVM 等）再使用针对当前问题收集到的数据进行学习。但是，如果当前我们面对的问题存在大量的学习数据，则也可以考虑从零开始同时进行单词的分布式表示和机器学习系统的学习。

下面让我们结合具体的例子来说明一下单词的分布式表示的使用方法。假设你现在开发并维护着一个拥有超过 1 亿用户的智能手机应用，你的公司每天都要收到堆积如山的用户邮件（在 Twitter 等上也有很多吐槽）。虽然有一部意见是积极的，但是也存在很多表达不满的用户意见。

为此，你考虑开发一个可以对用户发来的邮件（吐槽等）自动进行分类的系统。如图 4-22 所示，你想根据邮件的内容将用户情感分为 3 类。如果可以正确地对用户情感进行分类，就可以按序浏览表达不满的用户邮件。如此一来，或许可以发现应用的致命问题，并尽早采取应对措施，从而提高用户的满意度。



图 4-22　邮件的自动分类系统（情感分析）的例子

要开发邮件自动分类系统，首先需要从收集数据（邮件）开始。在这个例子中，我们收集用户发送的邮件，并人工对邮件进行标注，打上表示 3 类情感的标签（positive/neutral/negative）。标注工作结束后，用学习好的 word2vec 将邮件转化为向量。然后，将向量化的邮件及其情感标签输入某个情感分类系统（SVM 或神经网络等）进行学习。

如本例所示，可以基于单词的分布式表示将自然语言处理问题转化为向量，这样就可以利用常规的机器学习方法来解决问题。另外，这样也能从 word2vec 的迁移学习中受益。换句话说，利用 word2vec 的单词的分布式表示，可以期待大多数自然语言处理任务获得精度上的提高。

4.4.2 单词向量的评价方法

使用 word2vec，我们得到了单词的分布式表示。那么，我们应该如何评价分布式表示的优劣呢？本节我们将简单说明分布式表示的评价方法。

正如上述情感分析的例子那样，在现实世界中，单词的分布式表示往往被用在具体的应用中。我们最终想要的是一个高精度的系统。这里我们必须考虑到的是，这个系统（比如情感分析系统）是由多个子系统组成的。所谓多个子系统，拿刚才的例子来说，包括生成单词的分布式表示的系统（word2vec）、对特定问题进行分类的系统（比如进行情感分类的 SVM 等）。

单词的分布式表示的学习和分类系统的学习有时可能会分开进行。在这种情况下，如果要调查单词的分布式表示的维数如何影响最终的精度，首先需要进行单词的分布式表示的学习，然后再利用这个分布式表示进行另一个机器学习系统的学习。换句话说，在进行两个阶段的学习之后，才能进行评价。在这种情况下，由于需要调试出对两个系统都最优的超参数，所以非常费时。

因此，单词的分布式表示的评价往往与实际应用分开进行。此时，经常使用的评价指标有“相似度”和“类推问题”。

单词相似度的评价通常使用人工创建的单词相似度评价集来评估。比如，cat 和 animal 的相似度是 8，cat 和 car 的相似度是 2……类似这样，用 0 ~ 10 的分数人工地对单词之间的相似度打分。然后，比较人给出的分数和 word2vec 给出的余弦相似度，考察它们之间的相关性。

类推问题的评价是指，基于诸如“king : queen = man : ?”这样的类推问题，根据正确率测量单词的分布式表示的优劣。比如，论文 [27] 中给出了一个类推问题的评价结果，其部分内容如图 4-23 所示。

模型	维数	语料库的大小	Semantics	Syntax	Total
CBOW	300	16 亿	16.1	52.6	36.1
skip-gram	300	10 亿	61	61	61
CBOW	300	60 亿	63.6	67.4	65.7
skip-gram	300	60 亿	73.0	66.0	69.1
CBOW	1000	60 亿	57.3	68.9	63.7
skip-gram	1000	60 亿	66.1	65.1	65.6

图 4-23　基于类推问题的单词向量的评价结果（参考论文 [27]）

在图 4-23 中，以 word2vec 的模型、单词的分布式表示的维数和语料库的大小为参数进行了比较实验，结果在右侧的 3 列中。图 4-23 的 Semantics 列显示的是推断单词含义的类推问题（像“king : queen = actor : actress”这样询问单词含义的问题）的正确率，Syntax 列是询问单词形态信息的问题，比如“bad : worst = good : best”。



由图 4-23 可知：

- 模型不同，精度不同（根据语料库选择最佳的模型）
- 语料库越大，结果越好（始终需要大数据）
- 单词向量的维数必须适中（太大会导致精度变差）

基于类推问题可以在一定程度上衡量“是否正确理解了单词含义或语法问题”。因此，在自然语言处理的应用中，能够高精度地解决类推问题的单词的分布式表示应该可以获得好的结果。但是，单词的分布式表示的优劣对目标应用贡献多少（或者有无贡献），取决于待处理问题的具体情况，比如应用的类型或语料库的内容等。也就是说，不能保证类推问题的评价高，目标应用的结果就一定好。这一点请一定注意。

4.5　小结

本章我们以 word2vec 的高速化为主题，对上一章的 CBOW 模型进行了改进。具体来说，我们实现了 Embedding 层，并引入了一种名为负采样的新方法。这一改进的背景是，原先的方法随着语料库词汇量的增加，计算量也按比例增加。

利用“部分”数据而不是“全部”数据，这是本章的一个重要话题。正如人不能全知全能一样，以当前的计算机性能，要处理所有的数据也是不现实的。相反，仅处理对我们有用的那一小部分数据会有更好的效果。本章我们仔细研究了基于这一思想的负采样技术。负采样通过仅关注部分单词实现了计算的高速化。

通过上一章和本章的介绍，关于 word2vec 的一系列话题就要进入尾声了。word2vec 对自然语言处理领域产生了很大的影响，基于它获得的单词的分布式表示被应用在了各种自然语言处理任务中。另外，不仅限于自然语言处理，word2vec 的思想还被应用在了语音、图像和视频等领域中。希望读者能切实理解本章所讲的 word2vec 的相关内容，这些知识在许多领域都能派上用场。

本章所学的内容

- Embedding 层保存单词的分布式表示，在正向传播时，提取单词 ID 对应的向量
- 因为 word2vec 的计算量会随着词汇量的增加而成比例地增加，所以最好使用近似计算来加速
- 负采样技术采样若干负例，使用这一方法可以将多分类问题转化为二分类问题进行处理
- 基于 word2vec 获得的单词的分布式表示内嵌了单词含义，在相似的上下文中使用的单词在单词向量空间上处于相近的位置
- word2vec 的单词的分布式表示的一个特性是可以基于向量的加减法运算来求解类推问题
- word2vec 的迁移学习能力非常重要，它的单词的分布式表示可以应用于各种各样的自然语言处理任务

第 5 章
RNN

只记得我在一个昏暗潮湿的地方喵喵地哭泣着。

——夏目漱石《我是猫》

到目前为止，我们看到的神经网络都是前馈型神经网络。前馈（feedforward）是指网络的传播方向是单向的。具体地说，先将输入信号传给下一层（隐藏层），接收到信号的层也同样传给下一层，然后再传给下一层……像这样，信号仅在一个方向上传播。

虽然前馈网络结构简单、易于理解，但是可以应用于许多任务中。不过，这种网络存在一个大问题，就是不能很好地处理时间序列数据（以下简称为“时序数据”）。更确切地说，单纯的前馈网络无法充分学习时序数据的性质（模式）。于是，RNN（Recurrent Neural Network，循环神经网络）便应运而生。

本章我们将指出前馈网络的问题，并介绍 RNN 如何很好地解决这些问题。然后，我们会详细解释 RNN 的结构，并用 Python 对其进行实现。

5.1 概率和语言模型

作为介绍 RNN 的准备，我们将首先复习上一章的 word2vec，然后使用概率描述自然语言相关的现象，最后介绍从概率视角研究语言的“语言模型”。

5.1.1 概率视角下的word2vec

我们先复习一下 word2vec 的 CBOW 模型。这里，我们来考虑由单词序列 $w_1, w_2, \cdots, w_T$ 表示的语料库，将第 t 个单词作为目标词，将它左右的（第 $t-1$ 个和第 $t+1$ 个）单词作为上下文。



在本书中，目标词是指中间的单词，上下文是指目标词周围的单词。

如图 5-1 所示，CBOW 模型所做的事情就是从上下文（w_{t-1} 和 w_{t+1}）预测目标词（w_t）。

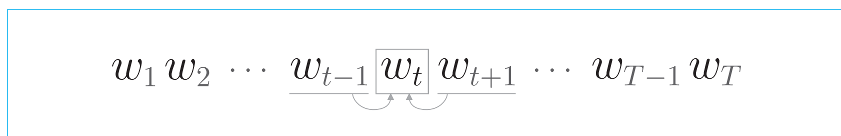


图 5-1 word2vec的CBOW模型：从上下文预测目标词

下面，我们用数学式来表示“当给定 w_{t-1} 和 w_{t+1} 时目标词是 w_t 的概率”，如式 (5.1) 所示：

$$P(w_t|w_{t-1}, w_{t+1}) \tag{5.1}$$

CBOW 模型对式 (5.1) 这一后验概率进行建模。这个后验概率表示“当给定 w_{t-1} 和 w_{t+1} 时 w_t 发生的概率”。这是窗口大小为 1 时的 CBOW 模型。

顺便提一下，我们之前考虑的窗口都是左右对称的。这里我们将上下文限定为左侧窗口，比如图 5-2 所示的情况。

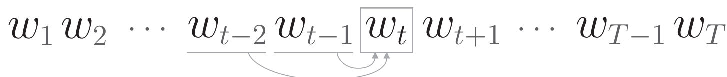


图 5-2　仅有左侧窗口的上下文

在仅将左侧 2 个单词作为上下文的情况下，CBOW 模型输出的概率如式 (5.2) 所示：

$$P(w_t|w_{t-2}, w_{t-1}) \tag{5.2}$$



word2vec的上下文窗口大小是超参数，可以设置为任何值。这里将窗口大小设置为左右非对称的形式，即左侧2个单词，右侧0个单词。这样设定的理由是提前考虑了后面要说的语言模型。

使用式 (5.2) 的写法，CBOW 模型的损失函数可以写成式 (5.3)。式 (5.3) 是从交叉熵误差推导出来的结果（具体请参考 1.3.1 节）。

$$L = -\log P(w_t|w_{t-2}, w_{t-1}) \tag{5.3}$$

CBOW 模型的学习旨在找到使式 (5.3) 表示的损失函数（确切地说，是整个语料库的损失函数之和）最小的权重参数。只要找到了这样的权重参数，CBOW 模型就可以更准确地从上下文预测目标词。

像这样，CBOW 模型的学习目的是从上下文预测出目标词。为了达成这一目标，随着学习的推进，（作为副产品）获得了编码了单词含义信息的单词的分布式表示。

那么，CBOW 模型本来的目的“从上下文预测目标词”是否可以用来做些什么呢？式 (5.2) 表示的概率 $P(w_t|w_{t-2}, w_{t-1})$ 是否可以在一些实际场景中发挥作用呢？说到这里，就要提一下语言模型了。

5.1.2 语言模型

语言模型（language model）给出了单词序列发生的概率。具体来说，就是使用概率来评估一个单词序列发生的可能性，即在多大程度上是自然的单词序列。比如，对于“you say goodbye”这一单词序列，语言模型给出高概率（比如 0.092）；对于“you say good die”这一单词序列，模型则给出低概率（比如 0.000 000 000 003 2）。

语言模型可以应用于多种应用，典型的例子有机器翻译和语音识别。比如，语音识别系统会根据人的发言生成多个句子作为候选。此时，使用语言模型，可以按照“作为句子是否自然”这一基准对候选句子进行排序。

语言模型也可以用于生成新的句子。因为语言模型可以使用概率来评价单词序列的自然程度，所以它可以根据这一概率分布造出（采样）单词。另外，第 7 章中我们会讨论如何使用语言模型生成文章。

现在，我们使用数学式来表示语言模型。这里考虑由 m 个单词 $w_1, \cdots, w_m$ 构成的句子，将单词按 $w_1, \cdots, w_m$ 的顺序出现的概率记为 $P(w_1, \cdots, w_m)$。因为这个概率是多个事件一起发生的概率，所以称为联合概率。

使用后验概率可以将这个联合概率 $P(w_1, \cdots, w_m)$ 分解成如下形式：

$$\begin{aligned} P(w_1, \cdots, w_m) &= P(w_m \mid w_1, \cdots, w_{m-1}) P(w_{m-1} \mid w_1, \cdots, w_{m-2}) \\ &\quad \cdots P(w_3 \mid w_1, w_2) P(w_2 \mid w_1) P(w_1) \\ &= \prod_{t=1}^{m} P(w_t \mid w_1, \cdots, w_{t-1}) \end{aligned} \tag{5.4}$$

①

与表示总和的 $\sum$（sigma）相对，式 (5.4) 中的 $\prod$（pi）表示所有元素相乘的乘积。如式 (5.4) 所示，联合概率可以由后验概率的乘积表示。

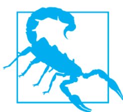
式 (5.4) 的结果可以从概率的乘法定理推导出来。这里我们花一点时间来说明一下乘法定理，并看一下式 (5.4) 的推导过程。

① 为了简化数学式，这里将 $P(w_1 \mid w_0)$ 作为 $P(w_1)$ 处理。

首先，概率的乘法定理可由下式表示：

$$P(A,B) = P(A|B)P(B) \tag{5.5}$$

式 (5.5) 表示的乘法定理是概率论中最重要的定理，意思是"A 和 B 两个事件共同发生的概率 $P(A, B)$"是"B 发生的概率 $P(B)$"和"B 发生后 A 发生的概率 $P(A|B)$"的乘积（这个解释感觉上非常自然）。



概率 $P(A, B)$ 也可以分解为 $P(A, B) = P(B|A)P(A)$。也就是说，根据将 A 和 B 中的哪一个作为后验概率的条件，存在 $P(A, B) = P(B|A)P(A)$ 和 $P(A, B) = P(A|B)P(B)$ 两种表示方法。

使用这个乘法定理，m 个单词的联合概率 $P(w_1, \cdots, w_m)$ 就可以用后验概率来表示。为了便于理解，我们先将式子如下变形：

$$P(\underbrace{w_1, \cdots, w_{m-1}}_{A}, w_m) = P(A, w_m) = P(w_m|A)P(A) \tag{5.6}$$

这里，将 $w_1, \cdots, w_{m-1}$ 整体表示为 A。这样一来，按照乘法定理，可以推导出式 (5.6)。接着，再对 $A(w_1, \cdots, w_{m-1})$ 进行同样的变形：

$$P(A) = P(\underbrace{w_1, \cdots, w_{m-2}}_{A'}, w_{m-1}) = P(A', w_{m-1}) = P(w_{m-1}|A')P(A') \tag{5.7}$$

像这样，单词序列每次减少一个，分解为后验概率。然后，重复这一过程，就可以推导出式 (5.4)。

如式 (5.4) 所示，联合概率 $P(w_1, \cdots, w_m)$ 可以表示为后验概率的乘积 $\prod P(w_t|w_1, \cdots, w_{t-1})$。这里需要注意的是，这个后验概率是以目标词左侧的全部单词为上下文（条件）时的概率，如图 5-3 所示。

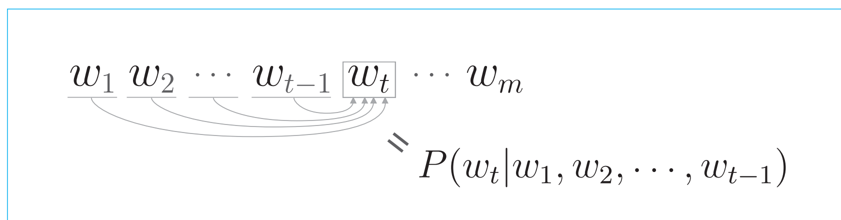


图5-3 语言模型中的后验概率：若以第t个单词为目标词，则第t个单词左侧的全部单词构成上下文（条件）

这里我们来总结一下，我们的目标是求 $P(w_t|w_1, \cdots, w_{t-1})$ 这个概率。如果能计算出这个概率，就能求得语言模型的联合概率 $P(w_1, \cdots, w_m)$。



由 $P(w_t|w_1, \cdots, w_{t-1})$ 表示的模型称为条件语言模型（conditional language model），有时也将其称为语言模型。

5.1.3 将CBOW模型用作语言模型？

那么，如果要把 word2vec 的 CBOW 模型（强行）用作语言模型，该怎么办呢？可以通过将上下文的大小限制在某个值来近似实现，用数学式可以如下表示：

$$P(w_1, \cdots, w_m) = \prod_{t=1}^{m} P(w_t|w_1, \cdots, w_{t-1}) \approx \prod_{t=1}^{m} P(w_t|w_{t-2}, w_{t-1}) \tag{5.8}$$

这里，我们将上下文限定为左侧的 2 个单词。如此一来，就可以用 CBOW 模型（CBOW 模型的后验概率）近似表示。



在机器学习和统计学领域，经常会听到“马尔可夫性”（或者“马尔可夫模型”“马尔可夫链”）这个词。马尔可夫性是指未来的状态仅依存于当前状态。此外，当某个事件的概率仅取决于其前面的N个事件时，称为“N阶马尔可夫链”。这里展示的是下一个单词仅取决于前面2个单词的模型，因此可以称为“2阶马尔可夫链”。

式 (5.8) 是使用 2 个单词作为上下文的例子，但是这个上下文的大小可以设定为任意长度（比如 5 或 10）。不过，虽说可以设定为任意长度，但必须是某个“固定”长度。比如，即便是使用左侧 10 个单词作为上下文的 CBOW 模型，其上下文更左侧的单词的信息也会被忽略，而这会导致问题，如图 5-4 中的例子所示。

Tom was watching TV in his room. Mary came into the room. Mary said hi to [?]

图 5-4　需要较长的上下文的问题示例：“？”中应填入什么单词？

在图 5-4 的问题中，“Tom 在房间看电视，Mary 进了房间”。根据该语境（上下文），正确答案应该是 Mary 向 Tom（或者“him”）打招呼。这里要获得正确答案，就必须将“?”前面第 18 个单词处的 Tom 记住。如果 CBOW 模型的上下文大小是 10，则这个问题将无法被正确回答。

那么，是否可以通过增大 CBOW 模型的上下文大小（比如变为 20 或 30）来解决此问题呢？的确，CBOW 模型的上下文大小可以任意设定，但是 CBOW 模型还存在忽视了上下文中单词顺序的问题。



CBOW 是 Continuous Bag-Of-Words 的简称。Bag-Of-Words 是“一袋子单词”的意思，这意味着袋子中单词的顺序被忽视了。

关于上下文的单词顺序被忽视这个问题，我们举个例子来具体说明。比如，在上下文是 2 个单词的情况下，CBOW 模型的中间层是那 2 个单词向量的和，如图 5-5 所示。

如图 5-5 的左图所示，在 CBOW 模型的中间层求单词向量的和，因此上下文的单词顺序会被忽视。比如，(you, say) 和 (say, you) 会被作为相同的内容进行处理。

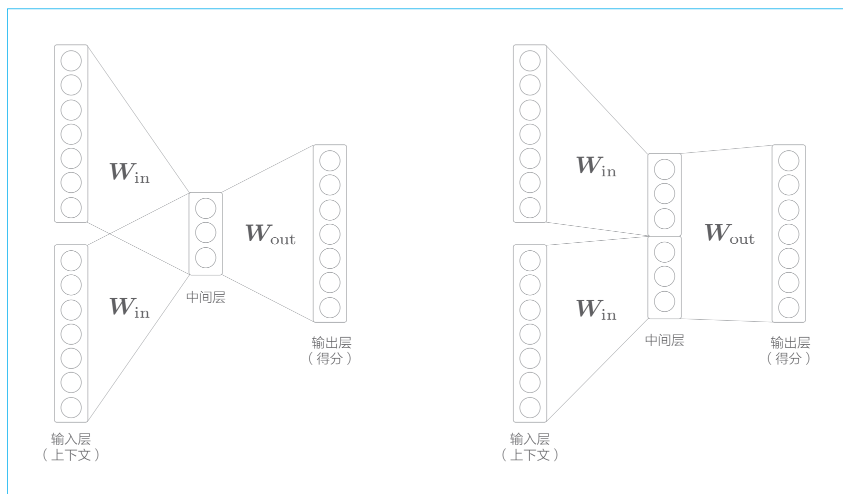


图5-5 左图是常规的CBOW模型。右图模型的中间层由上下文的各个单词向量“拼接”而成（图中的输入层是one-hot向量）

我们想要的是考虑了上下文中单词顺序的模型。为此，可以像图 5-5 中的右图那样，在中间层 “拼接”（concatenate）上下文的单词向量。实际上，“Neural Probabilistic Language Model” [28] 中提出的模型就采用了这个方法（关于模型的详细信息，请参考论文 [28]）。但是，如果采用拼接的方法，权重参数的数量将与上下文大小成比例地增加。显然，这是我们不愿意看到的。

那么，如何解决这里提出的问题呢？这就轮到 RNN 出场了。RNN 具有一个机制，那就是无论上下文有多长，都能将上下文信息记住。因此，使用 RNN 可以处理任意长度的时序数据。下面，我们就来感受一下 RNN 的魅力。



word2vec是以获取单词的分布式表示为目的的方法，因此一般不会用于语言模型。这里，为了引出RNN的魅力，我们拓展了话题，强行将word2vec的CBOW模型应用在了语言模型上。word2vec和基于RNN的语言模型是由托马斯 · 米科洛夫团队分别在2013年和2010年提出的。基于RNN的语言模型虽然也能获得单词的分布式表示，但是为了应对词汇量的增加、提高分布式表示的质量，word2vec被提了出来。

5.2 RNN

RNN（Recurrent Neural Network）中的 Recurrent 源自拉丁语，意思是“反复发生”，可以翻译为“重复发生”“周期性地发生”“循环”，因此 RNN 可以直译为“复发神经网络”或者“循环神经网络”。下面，我们将探讨“循环”一词。



Recurrent Neural Network 通常译为“循环神经网络”。另外，还有一种被称为 Recursive Neural Network（递归神经网络）的网络。这个网络主要用于处理树结构的数据，和循环神经网络不是一个东西。

5.2.1 循环的神经网络

“循环”是什么意思呢？是“反复并持续”的意思。从某个地点出发，经过一定时间又回到这个地点，然后重复进行，这就是“循环”一词的含义。这里要注意的是，循环需要一个“环路”。

只有存在了“环路”或者“回路”这样的路径，媒介（或者数据）才能在相同的地点之间来回移动。随着数据的循环，信息不断被更新。



血液在我们体内循环。今天流动的血液是接着昨天的血液继续流动的。另外，它也是接着一周前的、一个月前的、一年前的，甚至刚出生时的血液继续流动的。血液通过在体内循环，从过去一直被“更新”到现在。

RNN 的特征就在于拥有这样一个环路（或回路）。这个环路可以使数据不断循环。通过数据的循环，RNN 一边记住过去的数据，一边更新到最新的数据。

下面，我们来具体地看一下 RNN。这里，我们将 RNN 中使用的层称为“RNN 层”，如图 5-6 所示。

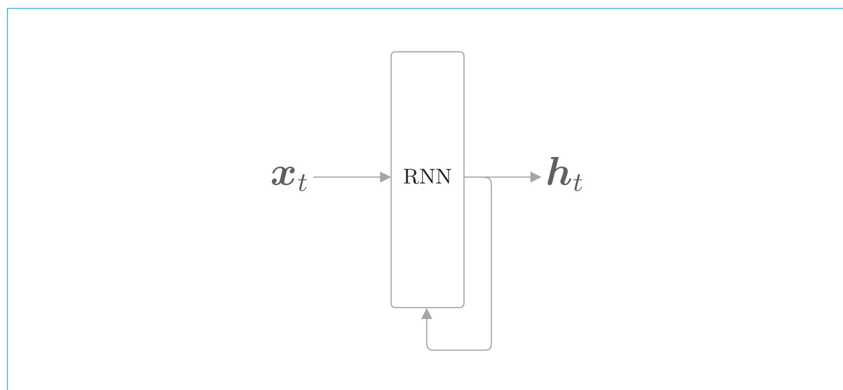


图5-6 拥有环路的RNN层

如图 5-6 所示，RNN 层有环路。通过该环路，数据可以在层内循环。在图 5-6 中，时刻 t 的输入是 $\boldsymbol{x}_t$，这暗示着时序数据 $(\boldsymbol{x}_0, \boldsymbol{x}_1, \cdots, \boldsymbol{x}_t, \cdots)$ 会被输入到层中。然后，以与输入对应的形式，输出 $(\boldsymbol{h}_0, \boldsymbol{h}_1, \cdots, \boldsymbol{h}_t, \cdots)$。

这里假定在各时刻向 RNN 层输入的 $\boldsymbol{x}_t$ 是向量。比如，在处理句子（单词序列）的情况下，将各个单词的分布式表示（单词向量）作为 $\boldsymbol{x}_t$ 输入 RNN 层。



仔细看一下图 5-6，可以发现输出有两个分叉，这意味着同一个东西被复制了。输出中的一个分叉将成为其自身的输入。

接着，我们来详细介绍一下图 5-6 的循环结构。在此之前，我们先将 RNN 层的绘制方法更改如下。

如图 5-7 所示，到目前为止，我们在绘制层时都是假设数据从左向右流动的。不过，从现在开始，为了节省纸面空间，我们将假设数据是从下向上流动的（这是为了在之后需要展开循环时，能够在左右方向上将层铺开）。

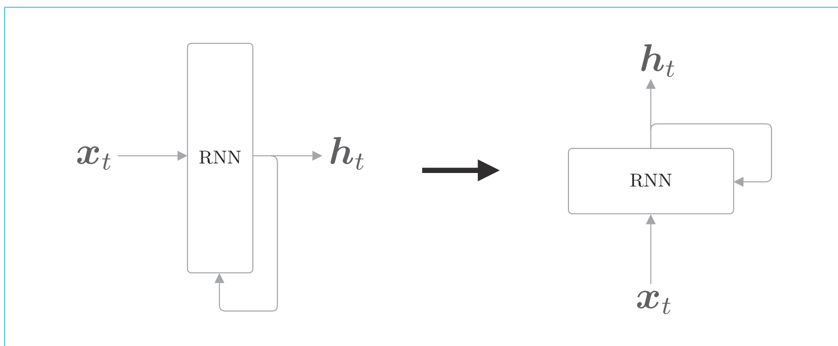


图 5-7 把层旋转 90 度

5.2.2 展开循环

现在，准备工作已经完成了，我们来仔细看一下 RNN 层的循环结构。RNN 的循环结构在之前的神经网络中从未出现过，但是通过展开循环，可以将其转化为我们熟悉的神经网络。百闻不如一见，现在我们就实际地进行展开（图 5-8）。

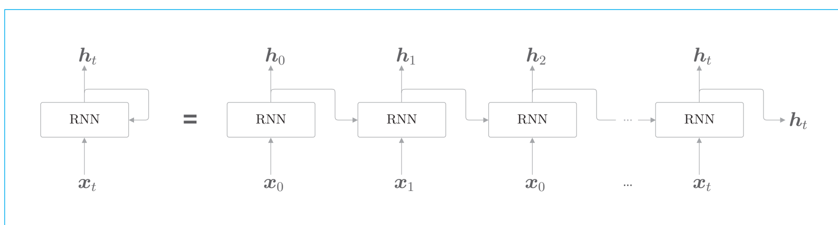


图 5-8 RNN 层的循环的展开

如图 5-8 所示，通过展开 RNN 层的循环，我们将其转化为了从左向右延伸的长神经网络。这和我们之前看到的前馈神经网络的结构相同（前馈网络的数据只向一个方向传播）。不过，图 5-8 中的多个 RNN 层都是“同一个层”，这一点与之前的神经网络是不一样的。



时序数据按时间顺序排列。因此，我们用“时刻”这个词指代时序数据的索引（比如，时刻t的输入数据为$\boldsymbol{x}_t$）。在自然语言处理的情况下，既使用“第t个单词”“第t个RNN层”这样的表述，也使用“时刻t的单词”或者“时刻t的RNN层”这样的表述。

由图 5-8 可以看出，各个时刻的 RNN 层接收传给该层的输入和前一个 RNN 层的输出，然后据此计算当前时刻的输出，此时进行的计算可以用下式表示：

$$\boldsymbol{h}_t = \tanh(\boldsymbol{h}_{t-1}\boldsymbol{W}_h + \boldsymbol{x}_t\boldsymbol{W}_x + \boldsymbol{b}) \tag{5.9}$$

首先说明一下式 (5.9) 中的符号。RNN 有两个权重，分别是将输入 $\boldsymbol{x}$ 转化为输出 $\boldsymbol{h}$ 的权重 $\boldsymbol{W}_x$ 和将前一个 RNN 层的输出转化为当前时刻的输出的权重 $\boldsymbol{W}_h$。此外，还有偏置 $\boldsymbol{b}$。这里，$\boldsymbol{h}_{t-1}$ 和 $\boldsymbol{x}_t$ 都是行向量。

在式 (5.9) 中，首先执行矩阵的乘积计算，然后使用 tanh 函数（双曲正切函数）变换它们的和，其结果就是时刻 t 的输出 $\boldsymbol{h}_t$。这个 $\boldsymbol{h}_t$ 一方面向上输出到另一个层，另一方面向右输出到下一个 RNN 层（自身）。

观察式 (5.9) 可以看出，现在的输出 $\boldsymbol{h}_t$ 是由前一个输出 $\boldsymbol{h}_{t-1}$ 计算出来的。从另一个角度看，这可以解释为，RNN 具有“状态”$\boldsymbol{h}$，并以式 (5.9) 的形式被更新。这就是说RNN 层是“具有状态的层”或“具有存储（记忆）的层”的原因。



RNN的$\boldsymbol{h}$存储“状态”，时间每前进一步（一个单位），它就以式(5.9)的形式被更新。许多文献中将RNN的输出$\boldsymbol{h}_t$称为隐藏状态（hidden state）或隐藏状态向量（hidden state vector），本书中也是如此。

另外，许多文献中将展开后的 RNN 层绘制成图 5-9 的左图。

在图 5-9 的左图中，从 RNN 层输出了两个箭头，但是请注意这两个箭头代表的是同一份数据（准确地说，是同一份数据被复制了）。在本书中，和之前一样，我们明确地在图中显示了输出处存在分叉，如图 5-9 的右图所示。

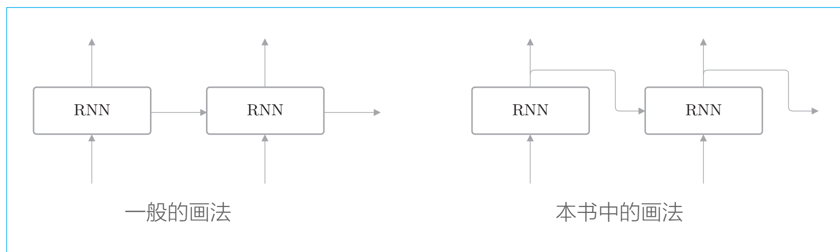


图 5-9 展开后的 RNN 层的画法比较

5.2.3 Backpropagation Through Time

将 RNN 层展开后，就可以视为在水平方向上延伸的神经网络，因此 RNN 的学习可以用与普通神经网络的学习相同的方式进行，如图 5-10 所示。

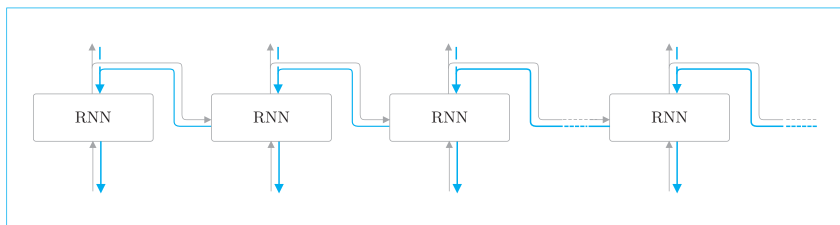


图 5-10 将循环展开后的 RNN 层的误差反向传播法

如图 5-10 所示，将循环展开后的 RNN 可以使用（常规的）误差反向传播法。换句话说，可以通过先进行正向传播，再进行反向传播的方式求目标梯度。因为这里的误差反向传播法是“按时间顺序展开的神经网络的误差反向传播法”，所以称为 Backpropagation Through Time（基于时间的反向传播），简称 BPTT。

通过该 BPTT，RNN 的学习似乎可以进行了，但是在这之前还有一个必须解决的问题，那就是学习长时序数据的问题。因为随着时序数据的时间跨度的增大，BPTT 消耗的计算机资源也会成比例地增大。另外，反向传播的梯度也会变得不稳定。



要基于BPTT求梯度，必须在内存中保存各个时刻的RNN层的中间数据（RNN层的反向传播将在后文中说明）。因此，随着时序数据变长，计算机的内存使用量（不仅仅是计算量）也会增加。

5.2.4 Truncated BPTT

在处理长时序数据时，通常的做法是将网络连接截成适当的长度。具体来说，就是将时间轴方向上过长的网络在合适的位置进行截断，从而创建多个小型网络，然后对截出来的小型网络执行误差反向传播法，这个方法称为 Truncated BPTT（截断的 BPTT）。



Truncated是“被截断”的意思。Truncated BPTT是指按适当长度截断的误差反向传播法。

在 Truncated BPTT 中，网络连接被截断，但严格地讲，只是网络的反向传播的连接被截断，正向传播的连接依然被维持，这一点很重要。也就是说，正向传播的信息没有中断地传播。与此相对，反向传播则被截断为适当的长度，以被截出的网络为单位进行学习。

现在，我们结合具体的例子来介绍 Truncated BPTT。假设有一个长度为 1000 的时序数据。在自然语言处理的情况下，这相当于一个有 1000 个单词的语料库。顺便说一下，我们之前处理的 PTB 数据集将多个串联起来的句子当作一个大的时序数据。这里也一样，将多个串联起来的句子当作一个时序数据。

在处理长度为 1000 的时序数据时，如果展开 RNN 层，它将成为在水平方向上排列有 1000 个层的网络。当然，无论排列多少层，都可以根据误差反向传播法计算梯度。但是，如果序列太长，就会出现计算量或者内存使用量方面的问题。此外，随着层变长，梯度逐渐变小，梯度将无法向前一层传递。因此，如图 5-11 所示，我们来考虑在水平方向上以适当的长度截断网络的反向传播的连接。

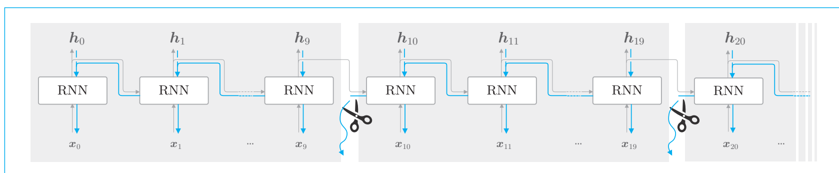


图5-11　在适当位置截断反向传播的连接。这里，将反向传播的连接中的某一段RNN层称为"块"（块的背景为灰色）

在图 5-11 中，我们截断了反向传播的连接，以使学习可以以 10 个 RNN 层为单位进行。像这样，只要将反向传播的连接截断，就不需要再考虑块范围以外的数据了，因此可以以各个块为单位（和其他块没有关联）完成误差反向传播法。

这里需要注意的是，虽然反向传播的连接会被截断，但是正向传播的连接不会。因此，在进行 RNN 的学习时，必须考虑到正向传播之间是有关联的，这意味着必须按顺序输入数据。下面，我们来说明什么是按顺序输入数据。



我们之前看到的神经网络在进行mini-batch学习时，数据都是随机选择的。但是，在RNN执行Truncated BPTT时，数据需要按顺序输入。

现在，我们考虑使用 Truncated BPTT 来学习 RNN。我们首先要做的是，将第 1 个块的输入数据 $(\boldsymbol{x}_0, \cdots, \boldsymbol{x}_9)$ 输入 RNN 层。这里要进行的处理如图 5-12 所示。

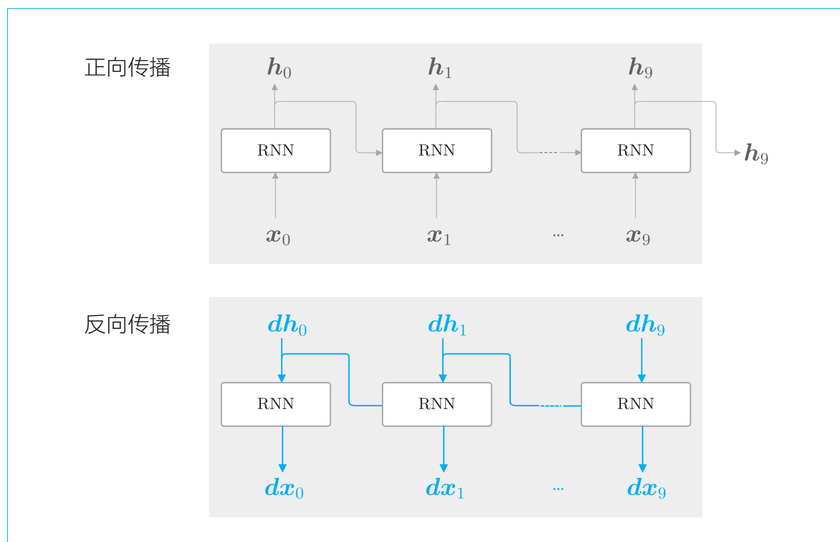


图5-12　第1个块的正向传播和反向传播：因为从后面的时刻传来的梯度被截断，所以误差反向传播法仅在本块内进行

如图 5-12 所示，先进行正向传播，再进行反向传播，这样可以得到所需的梯度。接着，对下一个块的输入数据 $(\boldsymbol{x}_{10}, \boldsymbol{x}_{11}, \cdots, \boldsymbol{x}_{19})$ 执行误差反向传播法，如图 5-13 所示。

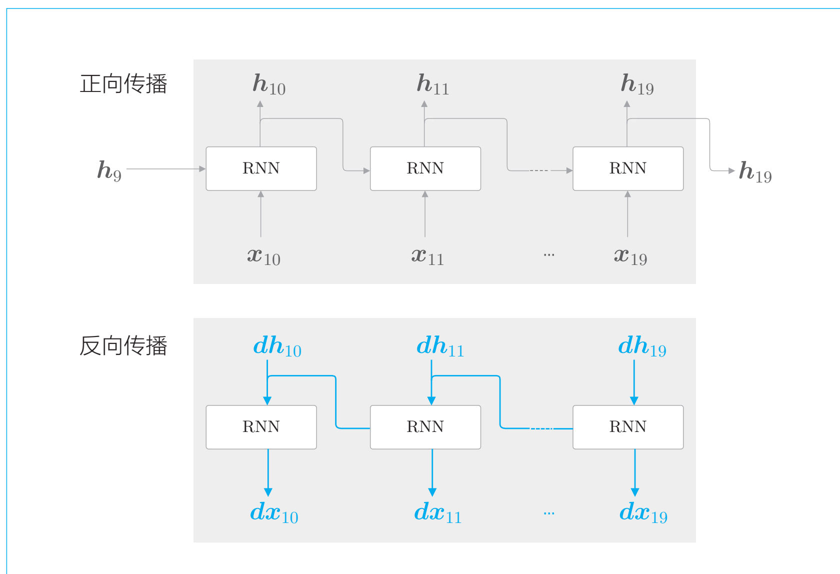


图 5-13　第 2 个块的正向传播和反向传播

这里，和第 1 个块一样，先执行正向传播，再执行反向传播。这里的重点是，这个正向传播的计算需要前一个块最后的隐藏状态 $\boldsymbol{h}_9$，这样可以维持正向传播的连接。

用同样的方法，继续学习第 3 个块，此时要使用第 2 个块最后的隐藏状态 $\boldsymbol{h}_{19}$。像这样，在 RNN 的学习中，通过将数据按顺序输入，从而继承隐藏状态进行学习。根据到目前为止的讨论，可知 RNN 的学习流程如图 5-14 所示。

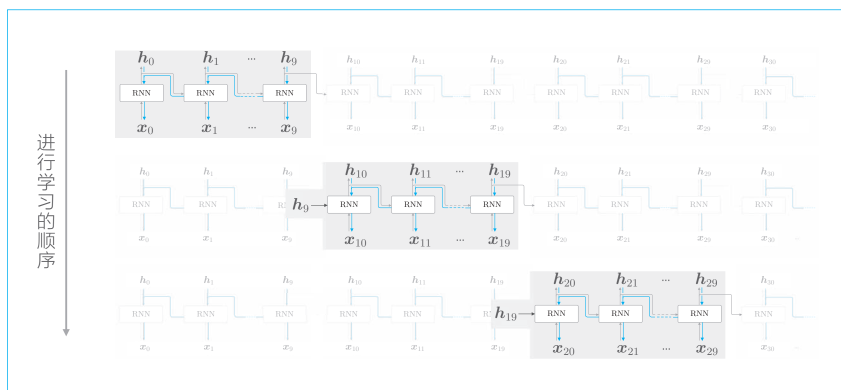


图5-14 Truncated BPTT的数据处理顺序（参见彩图）

如图 5-14 所示，Truncated BPTT 按顺序输入数据，进行学习。这样一来，能够在维持正向传播的连接的同时，以块为单位应用误差反向传播法。

5.2.5 Truncated BPTT的mini-batch学习

到目前为止，我们在探讨 Truncated BPTT 时，并没有考虑 mini-batch 学习。换句话说，我们之前的探讨对应于批大小为 1 的情况。为了执行 mini-batch 学习，需要考虑批数据，让它也能像图 5-14 一样按顺序输入数据。因此，在输入数据的开始位置，需要在各个批次中进行“偏移”。

为了说明“偏移”，我们仍用上一节的通过 Truncated BPTT 进行学习的例子，对长度为1000的时序数据，以时间长度10为单位进行截断。此时，如何将批大小设为 2 进行学习呢？在这种情况下，作为 RNN 层的输入数据，第 1 笔样本数据从头开始按顺序输入，第 2 笔数据从第 500 个数据开始按顺序输入。也就是说，将开始位置平移 500，如图 5-15 所示。

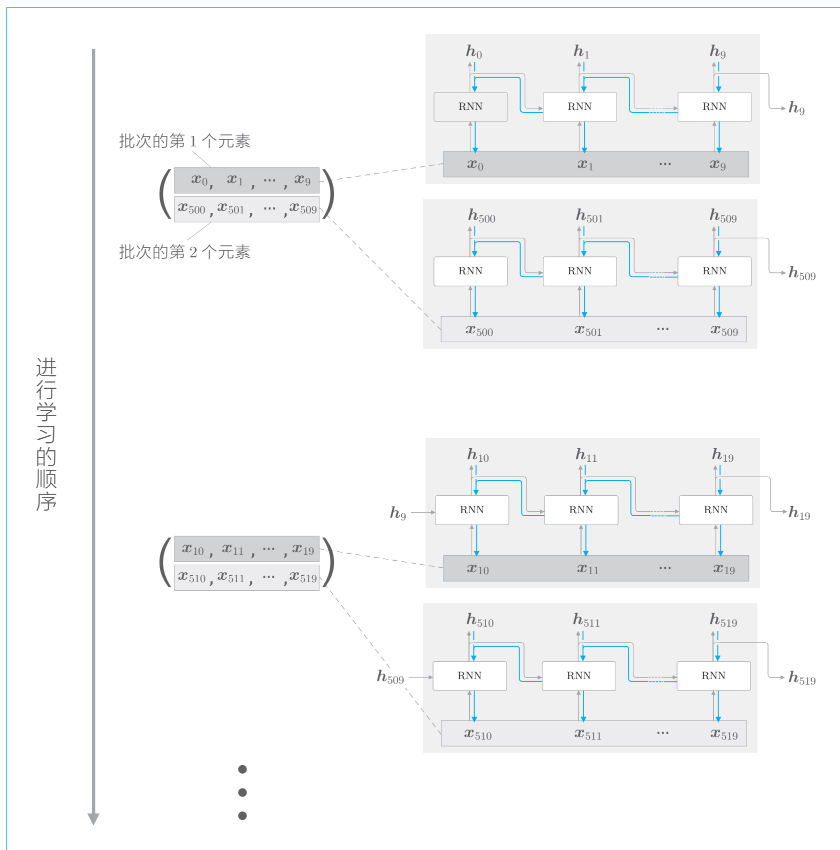


图 5-15　在进行 mini-batch 学习时，在各批次（各样本）中平移输入数据的开始位置

如图 5-15 所示，批次的第 1 个元素是 $\boldsymbol{x}_0, \cdots, \boldsymbol{x}_9$，批次的第 2 个元素是 $\boldsymbol{x}_{500}, \cdots, \boldsymbol{x}_{509}$，将这个 mini-batch 作为 RNN 的输入数据进行学习。因为要输入的数据是按顺序的，所以接下来是时序数据的第 10 ~ 19 个数据和第 510 ~ 519 个数据。像这样，在进行 mini-batch 学习时，平移各批次输入数据的开始位置，按顺序输入。此外，如果在按顺序输入数据的过程中遇到了结尾，则需要设法返回头部。

如上所述，虽然 Truncated BPTT 的原理非常简单，但是关于数据的输入方法有几个需要注意的地方。具体而言，一是要按顺序输入数据，二是

要平移各批次（各样本）输入数据的开始位置。这里的探讨有些复杂，大家一时间可能还不能理解，之后通过实际查看和运行代码，相信大家就能够理解了。

5.3　RNN 的实现

通过之前的探讨，我们已经看到了 RNN 的全貌。实际上，我们要实现的是一个在水平方向上延伸的神经网络。另外，考虑到基于 Truncated BPTT 的学习，只需要创建一个在水平方向上长度固定的网络序列即可，如图 5-16 所示。

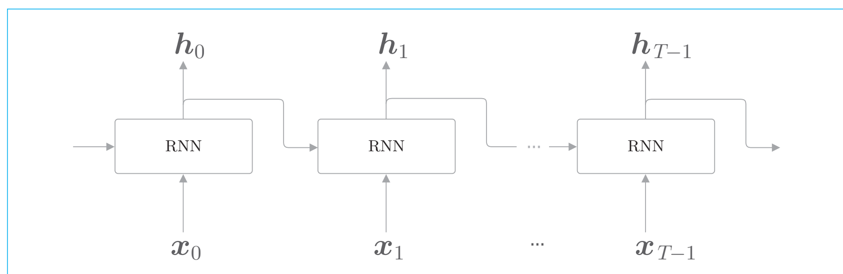


图 5-16　基于 RNN 的神经网络：在水平方向上长度固定

如图 5-16 所示，目标神经网络接收长度为 T 的时序数据（T 为任意值），输出各个时刻的隐藏状态 T 个。这里，考虑到模块化，将图 5-16 中在水平方向上延伸的神经网络实现为“一个层”，如图 5-17 所示。

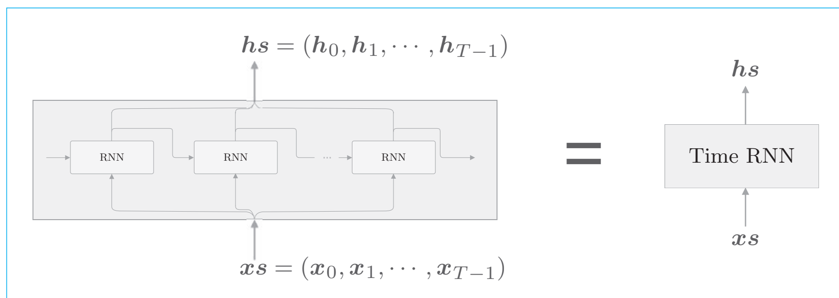


图 5-17 Time RNN 层：将展开循环后的层视为一个层

如图 5-17 所示，将垂直方向上的输入和输出分别捆绑在一起，就可以将水平排列的层视为一个层。换句话说，可以将 $(\boldsymbol{x}_0, \boldsymbol{x}_1, \cdots, \boldsymbol{x}_{T-1})$ 捆绑为 $\boldsymbol{xs}$ 作为输入，将 $(\boldsymbol{h}_0, \boldsymbol{h}_1, \cdots, \boldsymbol{h}_{T-1})$ 捆绑为 $\boldsymbol{hs}$ 作为输出。这里，我们将进行 Time RNN 层中的单步处理的层称为“RNN 层”，将一次处理 T 步的层称为“Time RNN 层”。



像 Time RNN 这样，将整体处理时序数据的层以单词“Time”开头命名，这是本书中规定的命名规范。之后，我们还会实现 Time Affine 层、Time Embedding 层等。

我们接下来进行的实现的流程是：首先，实现进行 RNN 单步处理的 `RNN` 类；然后，利用这个 `RNN` 类，完成一次进行 T 步处理的 `TimeRNN` 类。

5.3.1 RNN 层的实现

现在，我们来实现进行 RNN 单步处理的 `RNN` 类。首先复习一下 RNN 正向传播的数学式，如式 (5.10) 所示：

$$\boldsymbol{h}_t = \tanh(\boldsymbol{h}_{t-1}\boldsymbol{W}_h + \boldsymbol{x}_t\boldsymbol{W}_x + \boldsymbol{b}) \tag{5.10}$$

这里，我们将数据整理为 mini-batch 进行处理。因此，$\boldsymbol{x}_t$（和 $\boldsymbol{h}_t$）在行方向上保存各样本数据。在矩阵计算中，矩阵的形状检查非常重要。这里，假设批大小是 N，输入向量的维数是 D，隐藏状态向量的维数是 H，

则矩阵的形状检查可以像下面这样进行（图 5-18）。

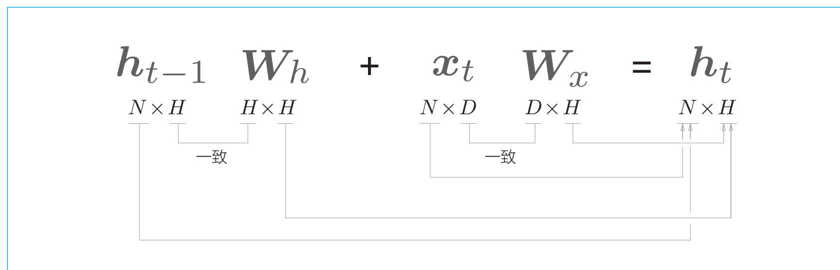


图5-18 形状检查：在矩阵乘积中，对应维度的元素数量要一致（这里省略偏置）

如图 5-18 所示，通过矩阵的形状检查，可以确认它的实现是否正确，至少可以确认它的计算是否成立。基于以上内容，现在我们给出 RNN 类的初始化方法和正向传播的 forward() 方法（✏ common/time_layers.py）。

```
class RNN:
    def __init__(self, Wx, Wh, b):
        self.params = [Wx, Wh, b]
        self.grads = [np.zeros_like(Wx), np.zeros_like(Wh),
↪   np.zeros_like(b)]
        self.cache = None

    def forward(self, x, h_prev):
        Wx, Wh, b = self.params
        t = np.dot(h_prev, Wh) + np.dot(x, Wx) + b
        h_next = np.tanh(t)

        self.cache = (x, h_prev, h_next)
        return h_next
```

RNN 的初始化方法接收两个权重参数和一个偏置参数。这里，将通过函数参数传进来的模型参数设置为列表类型的成员变量 params。然后，以各个参数对应的形状初始化梯度，并保存在 grads 中。最后，使用 None 对反向传播时要用到的中间数据 cache 进行初始化。

正向传播的 forward(x, h_prev) 方法接收两个参数：从下方输入的 x 和

从左边输入的 h_prev。剩下的就是按式 (5.10) 进行实现。顺便说一下，这里从前一个 RNN 层接收的输入是 h_prev，当前时刻的 RNN 层的输出（= 下一时刻的 RNN 层的输入）是 h_next。

接下来，我们继续实现 RNN 的反向传播。在此之前，让我们通过图 5-19 的计算图再次确认一下 RNN 的正向传播。

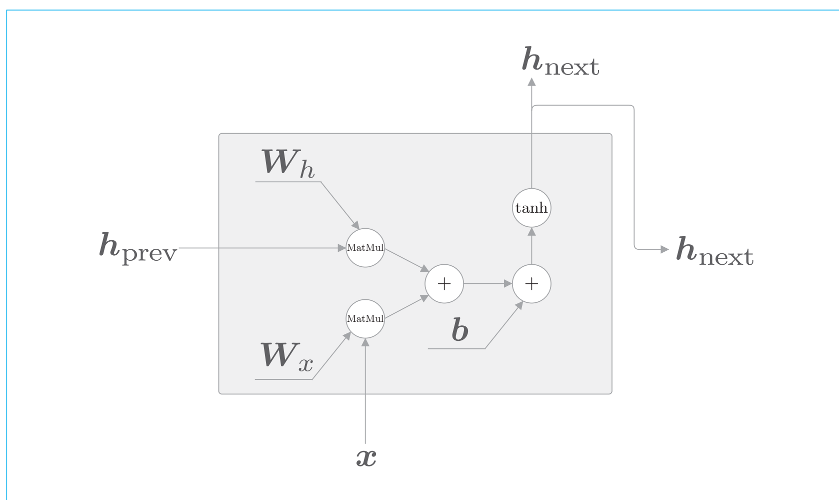


图 5-19　RNN 层的计算图（MatMul 节点表示矩阵乘积）

RNN 层的正向传播可由图 5-19 的计算图表示。这里进行的计算由矩阵乘积“MatMul”、加法“+”和“tanh”这 3 种运算构成。此外，因为偏置 $\boldsymbol{b}$ 的加法运算会触发广播操作，所以严格地讲，这里还应该加上 Repeat 节点。不过简单起见，这里省略了它（具体请参考 1.3.4.3 节）。

那么，图 5-19 的计算图的反向传播是什么样的呢？答案很简单。因为这 3 种运算的反向传播我们都已经掌握了（关于反向传播，请参考 1.3 节），剩下就是基于图 5-20，按正向传播的反方向实现各个运算的反向传播。

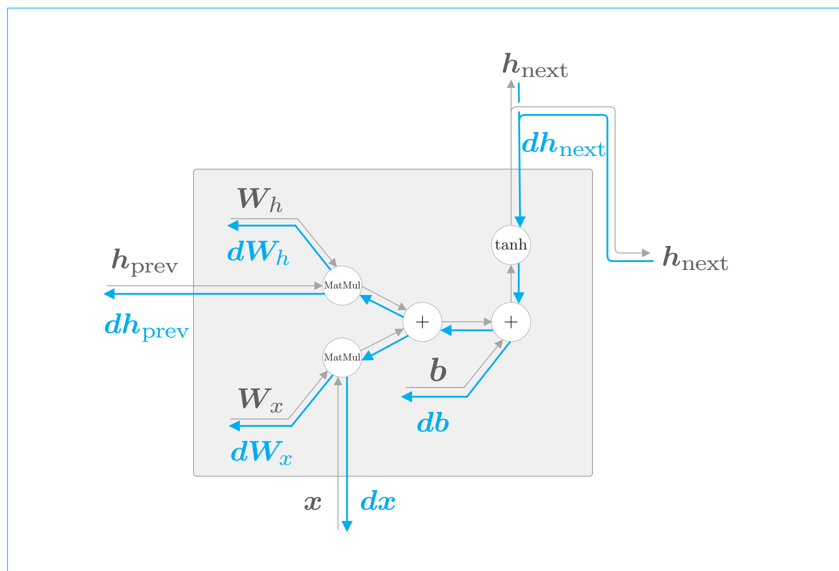


图 5-20 基于 RNN 层的计算图的反向传播

下面实现 RNN 层的 backward()。参考图 5-20，可以如下实现。

```
def backward(self, dh_next):
    Wx, Wh, b = self.params
    x, h_prev, h_next = self.cache

    dt = dh_next * (1 - h_next ** 2)
    db = np.sum(dt, axis=0)
    dWh = np.dot(h_prev.T, dt)
    dh_prev = np.dot(dt, Wh.T)
    dWx = np.dot(x.T, dt)
    dx = np.dot(dt, Wx.T)

    self.grads[0][...] = dWx
    self.grads[1][...] = dWh
    self.grads[2][...] = db

    return dx, dh_prev
```

以上就是 RNN 层的反向传播的实现。接下来，我们将实现 Time RNN 层。

5.3.2　Time RNN 层的实现

Time RNN 层由 T 个 RNN 层构成（T 可以设置为任意值），如图 5-21 所示。

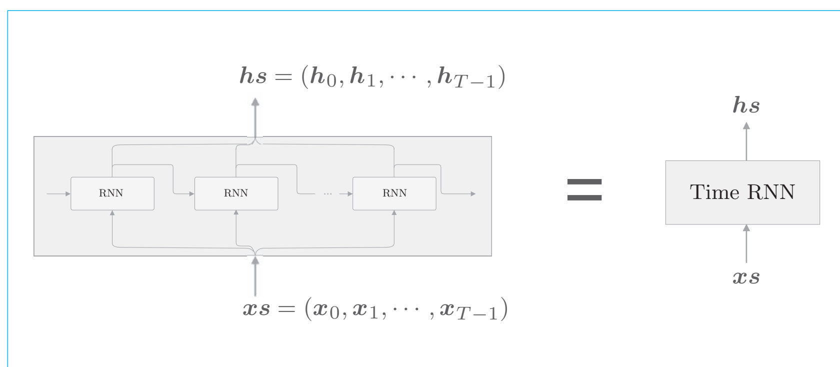


图 5-21　Time RNN 层和 RNN 层

由图 5-21 可知，Time RNN 层是 T 个 RNN 层连接起来的网络。我们将这个网络实现为 Time RNN 层。这里，RNN 层的隐藏状态 $\boldsymbol{h}$ 保存在成员变量中。如图 5-22 所示，在进行隐藏状态的"继承"时会用到它。

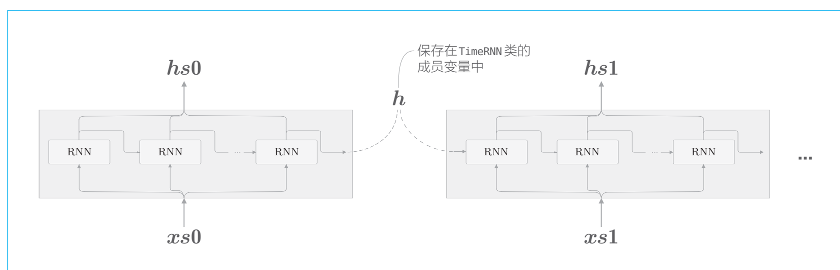


图 5-22　Time RNN 层将隐藏状态 $\boldsymbol{h}$ 保存在成员变量中，以在块之间继承隐藏状态

如图 5-22 所示，我们使用 Time RNN 层管理 RNN 层的隐藏状态。这样一来，使用 Time RNN 的人就不必考虑 RNN 层的隐藏状态的"继承工

作”了。另外，我们可以用 stateful 这个参数来控制是否继承隐藏状态。

下面，我们来看一下 Time RNN 层的实现。首先实现初始化方法和两个方法（✏ common/time_layers.py）。

```
class TimeRNN:
    def __init__(self, Wx, Wh, b, stateful=False):
        self.params = [Wx, Wh, b]
        self.grads = [np.zeros_like(Wx), np.zeros_like(Wh),
 ↪  np.zeros_like(b)]
        self.layers = None

        self.h, self.dh = None, None
        self.stateful = stateful

    def set_state(self, h):
        self.h = h

    def reset_state(self):
        self.h = None
```

初始化方法的参数有权重、偏置和布尔型（True/False）的 stateful。一个成员变量 layers 在列表中保存多个 RNN 层，另一个成员变量，h 保存调用 forward() 方法时的最后一个 RNN 层的隐藏状态。另外，在调用 backward() 时，dh 保存传给前一个块的隐藏状态的梯度（关于 dh，我们会在反向传播的实现中说明）。



考虑到 TimeRNN 类的扩展性，将设定 Time RNN 层的隐藏状态的方法实现为 set_state(h)。另外，将重设隐藏状态的方法实现为 reset_state()。

上述参数中的 stateful 是“有状态”的意思。在本书的实现中，当 stateful 为 True 时，Time RNN 层“有状态”。这里说的“有状态”是指维持 Time RNN 层的隐藏状态。也就是说，无论时序数据多长，Time RNN 层的正向传播都可以不中断地进行。而当 stateful 为 False 时，每次调用

Time RNN层的 forward() 时，第一个RNN层的隐藏状态都会被初始化为零矩阵（所有元素均为0的矩阵）。这是没有状态的模式，称为“无状态”。



在处理长时序数据时，需要维持RNN的隐藏状态，这一功能通常用“stateful”一词表示。在许多深度学习框架中，RNN层都有 stateful 参数，该参数用于指定是否保存上一时刻的隐藏状态。

接着，我们来看一下正向传播的实现。

```
def forward(self, xs):
    Wx, Wh, b = self.params
    N, T, D = xs.shape
    D, H = Wx.shape

    self.layers = []
    hs = np.empty((N, T, H), dtype='f')

    if not self.stateful or self.h is None:
        self.h = np.zeros((N, H), dtype='f')

    for t in range(T):
        layer = RNN(*self.params)
        self.h = layer.forward(xs[:, t, :], self.h)
        hs[:, t, :] = self.h
        self.layers.append(layer)

    return hs
```

正向传播的 forward(xs) 方法从下方获取输入 xs，xs 囊括了 T 个时序数据。因此，如果批大小是 N，输入向量的维数是 D，则 xs 的形状为 (N,T,D)。

在首次调用时（self.h 为 None 时），RNN层的隐藏状态 h 由所有元素均为0的矩阵初始化。另外，在成员变量 stateful 为 False 的情况下，h 将总是被重置为零矩阵。

在主体实现中，首先通过 hs=np.empty((N, T, H), dtype='f') 为输出准备一个“容器”。接着，在 T 次 for 循环中，生成RNN层，并将其添加到成

员变量 `layers` 中。然后，计算 RNN 层各个时刻的隐藏状态，并存放在 `hs` 的对应索引（时刻）中。



如果调用Time RNN层的`forward()`方法，则成员变量`h`中将存放最后一个RNN层的隐藏状态。在`stateful`为`True`的情况下，在下一次调用`forward()`方法时，刚才的成员变量`h`将被继续使用。而在`stateful`为`False`的情况下，成员变量`h`将被重置为零向量。

接下来是 Time RNN 层的反向传播的实现。用计算图绘制这个反向传播，如图 5-23 所示。

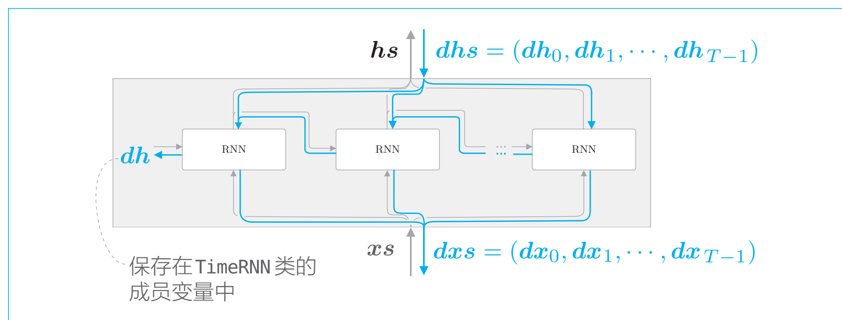


图 5-23　Time RNN 层的反向传播

在图 5-23 中，将从上游（输出侧的层）传来的梯度记为 $\boldsymbol{dhs}$，将流向下游的梯度记为 $\boldsymbol{dxs}$。因为这里我们进行的是 Truncated BPTT，所以不需要流向这个块上一时刻的反向传播。不过，我们将流向上一时刻的隐藏状态的梯度存放在成员变量 `dh` 中。这是因为在第 7 章探讨 seq2seq（sequence-to-sequence，序列到序列）时会用到它（具体请参考第 7 章）。

以上就是 Time RNN 层的反向传播的全貌图。如果关注第 t 个 RNN 层，则它的反向传播如图 5-24 所示。

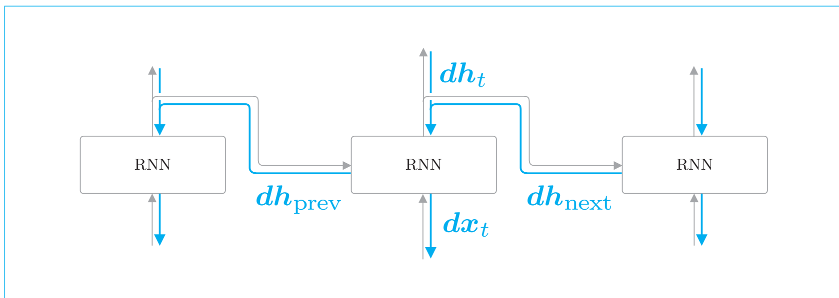


图 5-24 第 t 个 RNN 层的反向传播

从上方传来的梯度 $\boldsymbol{dh}_t$ 和从将来的层传来的梯度 $\boldsymbol{dh}_{\text{next}}$ 会传到第 t 个 RNN 层。这里需要注意的是，RNN 层的正向传播的输出有两个分叉。在正向传播存在分叉的情况下，在反向传播时各梯度将被求和。因此，在反向传播时，流向 RNN 层的是求和后的梯度。考虑到以上这些，反向传播的实现如下所示。

```
def backward(self, dhs):
    Wx, Wh, b = self.params
    N, T, H = dhs.shape
    D, H = Wx.shape

    dxs = np.empty((N, T, D), dtype='f')
    dh = 0
    grads = [0, 0, 0]
    for t in reversed(range(T)):
        layer = self.layers[t]
        dx, dh = layer.backward(dhs[:, t, :] + dh) # 求和后的梯度
        dxs[:, t, :] = dx

        for i, grad in enumerate(layer.grads):
            grads[i] += grad

    for i, grad in enumerate(grads):
        self.grads[i][...] = grad
    self.dh = dh

    return dxs
```

这里，首先创建传给下游的梯度的“容器”（dxs）。接着，按与正向传播相反的方向，调用 RNN 层的 `backward()` 方法，求得各个时刻的梯度 `dx`，并存放在 `dxs` 的对应索引处。另外，关于权重参数，需要求各个 RNN 层的权重梯度的和，并通过“...”用最终结果覆盖成员变量 `self.grads`。



在 Time RNN 层中有多个 RNN 层。另外，这些 RNN 层使用相同的权重。因此，Time RNN 层的（最终）权重梯度是各个 RNN 层的权重梯度之和。

以上就是对 Time RNN 层的实现的说明。

5.4 处理时序数据的层的实现

本章我们的目标是使用 RNN 实现语言模型。目前我们已经实现了 RNN 层和整体处理时序数据的 Time RNN 层，本节将创建几个可以处理时序数据的新层。我们将基于 RNN 的语言模型称为 RNNLM（RNN Language Model，RNN 语言模型）。现在，我们来完成 RNNLM。

5.4.1 RNNLM 的全貌图

首先，我们看一下 RNNLM 使用的网络。图 5-25 所示为最简单的 RNNLM 的网络，其中左图显示了 RNNLM 的层结构，右图显示了在时间轴上展开后的网络。

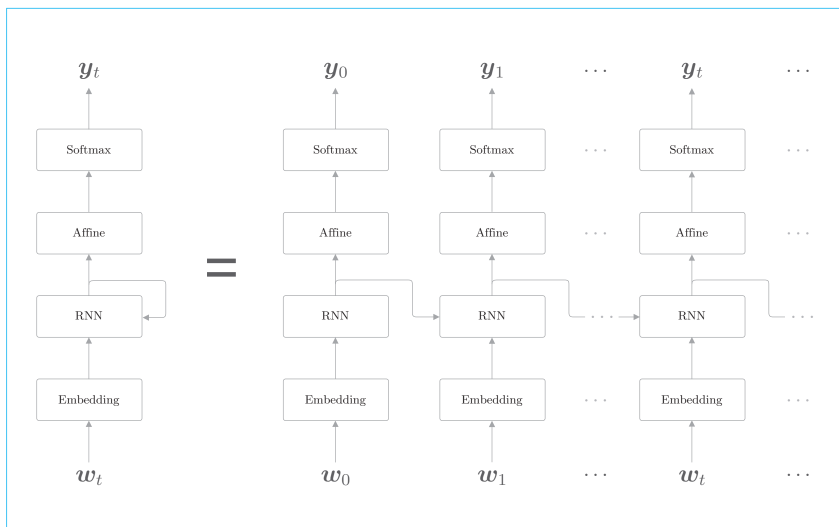


图 5-25　RNNLM 的网络图（左图是展开前，右图是展开后）

图 5-25 中的第 1 层是 Embedding 层，该层将单词 ID 转化为单词的分布式表示（单词向量）。然后，这个单词向量被输入到 RNN 层。RNN 层向下一层（上方）输出隐藏状态，同时也向下一时刻的 RNN 层（右侧）输出隐藏状态。RNN 层向上方输出的隐藏状态经过 Affine 层，传给 Softmax 层。

现在，我们仅考虑正向传播，向图 5-25 的神经网络传入具体的数据，并观察输出结果。这里使用的句子还是我们熟悉的 “you say goodbye and i say hello.”，此时 RNNLM 进行的处理如图 5-26 所示。

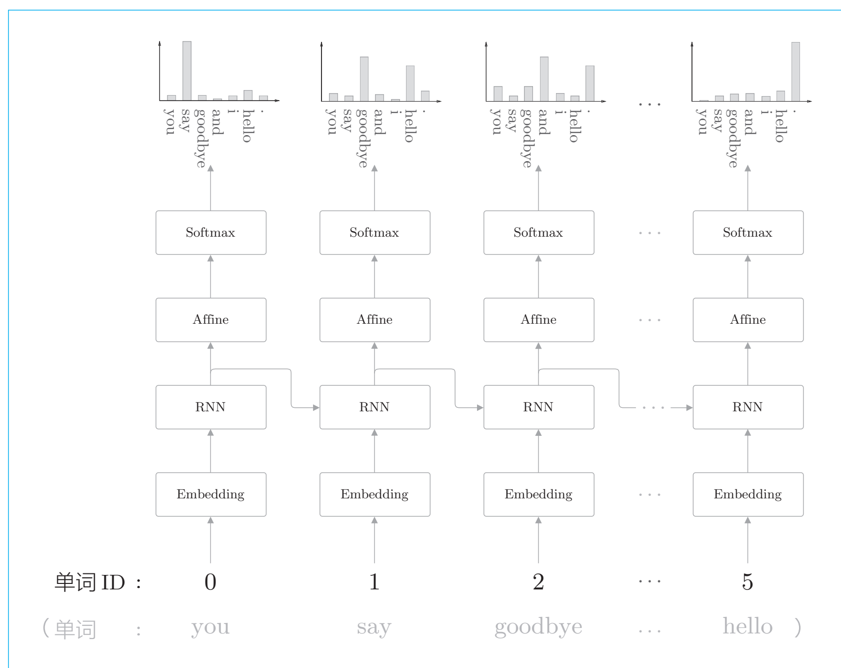


图5-26 处理样本语料库“you say goodbye and i say hello.”的RNNLM的例子

如图 5-26 所示，被输入的数据是单词 ID 列表。首先，我们关注第 1 个时刻。作为第 1 个单词，单词 ID 为 0 的 you 被输入。此时，查看 Softmax 层输出的概率分布，可知 say 的概率最高，这表明正确预测出了 you 后面出现的单词为 say。当然，这样的正确预测只在有“好的”（学习顺利的）权重时才会发生。

接着，我们关注第 2 个单词 say。此时，Softmax 层的输出在 goodbye 处和 hello 处概率较高。确实，“you say goodby”和“you say hello”都是很自然的句子（顺便说一下，正确答案是 goodbye）。这里需要注意的是，RNN 层“记忆”了“you say”这一上下文。更准确地说，RNN 将“you say”这一过去的信息保存为了简短的隐藏状态向量。RNN 层的工作是将这个信息传送到上方的 Affine 层和下一时刻的 RNN 层。

像这样，RNNLM 可以“记忆”目前为止输入的单词，并以此为基础预

测接下来会出现的单词。RNN 层通过从过去到现在继承并传递数据，使得编码和存储过去的信息成为可能。

5.4.2　Time 层的实现

之前我们将整体处理时序数据的层实现为了 Time RNN 层，这里也同样使用 Time Embedding 层、Time Affine 层等来实现整体处理时序数据的层。一旦创建了这些 Time ××层，我们的目标神经网络就可以像图 5-27 这样实现。

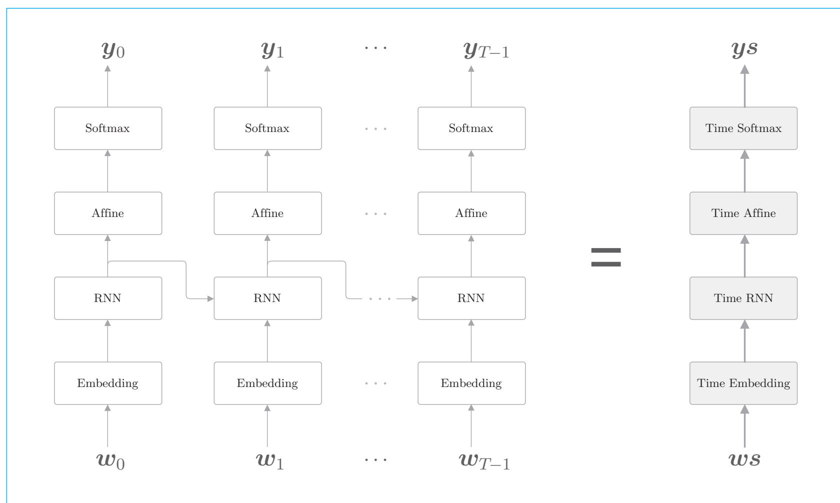


图 5-27　将整体处理时序数据的层实现为 Time ××层



我们将整体处理含有 T 个时序数据的层称为“Time ××层”。如果可以实现这些层，通过像组装乐高积木一样组装它们，就可以完成处理时序数据的网络。

Time 层的实现很简单。比如，在 Time Affine 层的情况下，只需要像图 5-28 那样，准备 T 个 Affine 层分别处理各个时刻的数据即可。

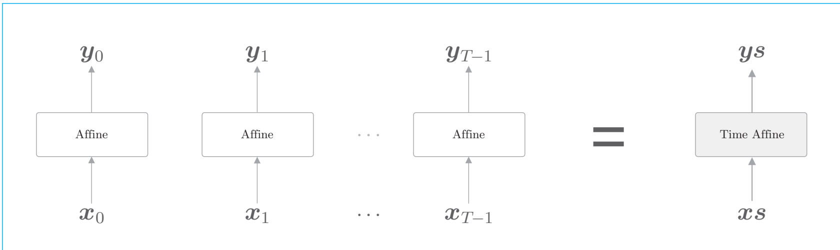


图5-28 将Time Affine层实现为T个Affine层的集合

Time Embedding 层也一样，在正向传播时准备 T 个 Embedding 层，由各个 Embedding 层处理各个时刻的数据。

关于 Time Affine 层和 Time Embedding 层没有什么特别难的内容，我们就不再赘述了。需要注意的是，Time Affine 层并不是单纯地使用 T 个 Affine 层，而是使用矩阵运算实现了高效的整体处理。感兴趣的读者可以参考源代码（`common/time_layers.py` 的 `TimeAffine` 类）。接下来我们看一下时序版本的 Softmax。

我们在 Softmax 中一并实现损失误差 Cross Entropy Error 层。这里，按照图 5-29 所示的网络结构实现 Time Softmax with Loss 层。

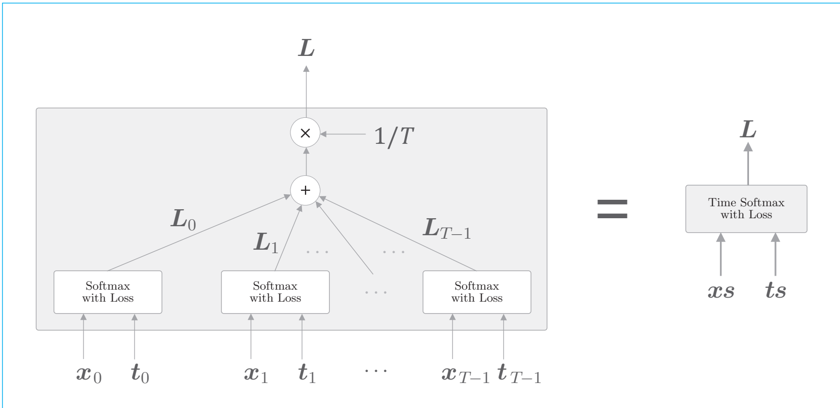


图5-29 Time Softmax with Loss层的全貌图

图 5-29 中的 $\boldsymbol{x}_0$、$\boldsymbol{x}_1$ 等数据表示从下方的层传来的得分（得分是正规

化为概率之前的值），$\boldsymbol{t}_0$、$\boldsymbol{t}_1$ 等数据表示正确解标签。如该图所示，T 个 Softmax with Loss 层各自算出损失，然后将它们加在一起取平均，将得到的值作为最终的损失。此处进行的计算可用下式表示：

$$L = \frac{1}{T}(L_0 + L_1 + \cdots + L_{T-1}) \tag{5.11}$$

顺便说一下，本书的Softmax with Loss层计算mini-batch的平均损失。具体而言，假设 mini-batch 有 N 笔数据，通过先求 N 笔数据的损失之和，再除以 N，可以得到单笔数据的平均损失。这里也一样，通过取时序数据的平均，可以求得单笔数据的平均损失作为最终的输出。

以上就是对 Time 层的说明。这里只是做了一个简短的说明，实际的实现可以在 `common/time_layers.py` 中找到，感兴趣的读者可以参考一下。

5.5　RNNLM 的学习和评价

实现 RNNLM 所需要的层都已经准备好了，现在我们来实现 RNNLM，并对其进行训练，然后再评价一下它的结果。

5.5.1　RNNLM 的实现

这里我们将 RNNLM 使用的网络实现为 `SimpleRnnlm` 类，其层结构如图 5-30 所示。

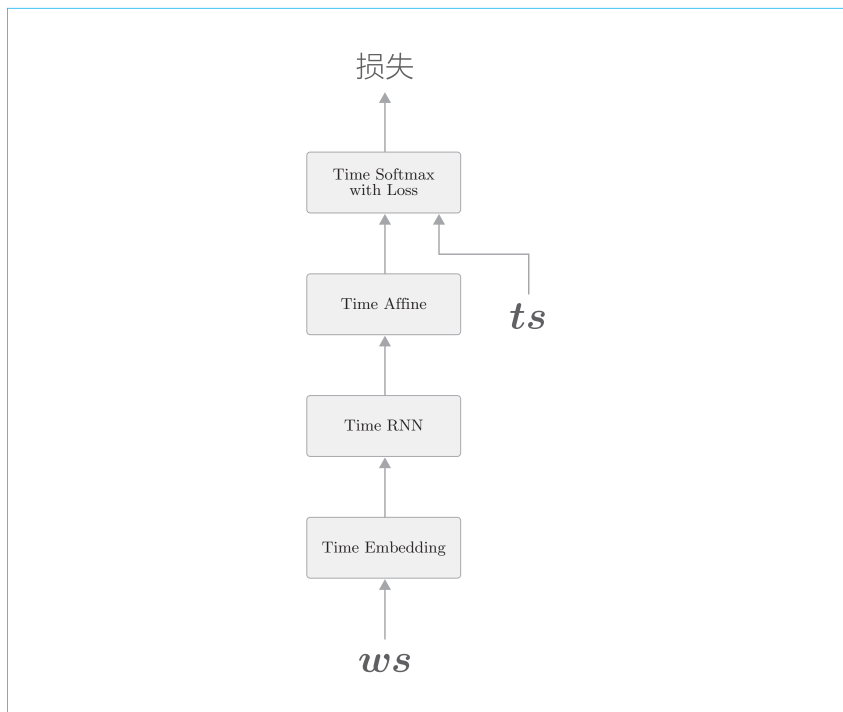


图5-30 SimpleRnnlm的层结构：RNN层的状态在类内部进行管理

如图 5-30 所示，`SimpleRnnlm` 类是一个堆叠了 4 个 Time 层的神经网络。我们先来看一下初始化的代码（✏ `ch05/simple_rnnlm.py`）。

```
import sys
sys.path.append('..')
import numpy as np
from common.time_layers import *

class SimpleRnnlm:
    def __init__(self, vocab_size, wordvec_size, hidden_size):
        V, D, H = vocab_size, wordvec_size, hidden_size
        rn = np.random.randn

        # 初始化权重
        embed_W = (rn(V, D) / 100).astype('f')
```

```
rnn_Wx = (rn(D, H) / np.sqrt(D)).astype('f')
rnn_Wh = (rn(H, H) / np.sqrt(H)).astype('f')
rnn_b = np.zeros(H).astype('f')
affine_W = (rn(H, V) / np.sqrt(H)).astype('f')
affine_b = np.zeros(V).astype('f')

# 生成层
self.layers = [
    TimeEmbedding(embed_W),
    TimeRNN(rnn_Wx, rnn_Wh, rnn_b, stateful=True),
    TimeAffine(affine_W, affine_b)
]
self.loss_layer = TimeSoftmaxWithLoss()
self.rnn_layer = self.layers[1]

# 将所有的权重和梯度整理到列表中
self.params, self.grads = [], []
for layer in self.layers:
    self.params += layer.params
    self.grads += layer.grads
```

这里，对各个层使用的参数（权重和偏置）进行初始化，生成必要的层。假设使用 Truncated BPTT 进行学习，将 Time RNN 层的 `stateful` 设置为 `True`，如此 Time RNN 层就可以继承上一时刻的隐藏状态。

另外，在上面的初始化代码中，RNN 层和 Affine 层使用了“Xavier 初始值”。如图 5-31 所示，在上一层的节点数是 n 的情况下，使用标准差为 $\frac{1}{\sqrt{n}}$ 的分布作为 Xavier 初始值[①]。顺便说一下，标准差可以直观地解释为表示数据分散程度的指标。

① 这是一个简略版的实现，原始论文中提出的权重初始值还考虑了下一层的节点数。

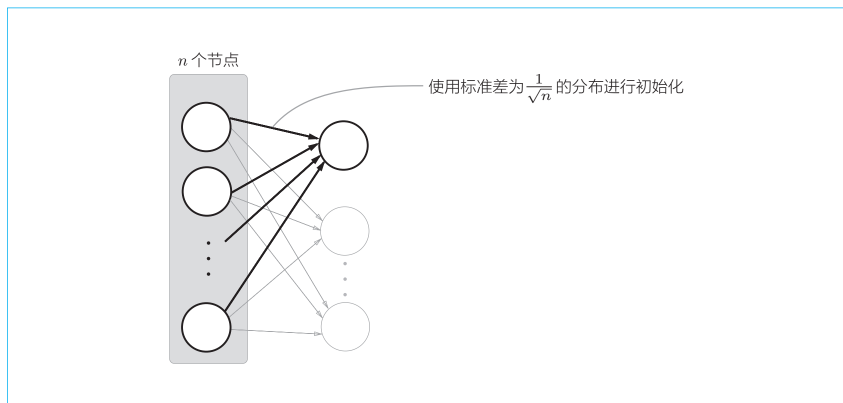


图 5-31 Xavier 初始值：在上一层有 n 个节点的情况下，使用标准差为 $\frac{1}{\sqrt{n}}$ 的分布作为初始值



在深度学习中，权重的初始值非常重要。关于这一点，我们在前作《深度学习入门：基于 Python 的理论与实现》中已经进行了详细的探讨。同样，对 RNN 而言，权重的初始值也很重要。通过设置好的初始值，学习的进展和最终的精度都会有很大变化。本书此后都将使用 Xavier 初始值作为权重的初始值。另外，在语言模型的相关研究中，经常使用 `0.01 * np.random.uniform(...)` 这样的经过缩放的均匀分布。

接着，我们来实现 `forward()` 方法、`backward()` 方法和 `reset_state()` 方法。

```
def forward(self, xs, ts):
    for layer in self.layers:
        xs = layer.forward(xs)
    loss = self.loss_layer.forward(xs, ts)
    return loss

def backward(self, dout=1):
    dout = self.loss_layer.backward(dout)
    for layer in reversed(self.layers):
        dout = layer.backward(dout)
    return dout
```

```
    def reset_state(self):
        self.rnn_layer.reset_state()
```

可以看出实现非常简单。在各个层中，正向传播和反向传播都正确地进行了实现。因此，我们只要以正确的顺序调用 `forward()`（或者 `backward()`）即可。方便起见，这里将重设网络状态的方法实现为 `reset_state()`。以上就是对 `SimpleRnnlm` 类的说明。

5.5.2 语言模型的评价

`SimpleRnnlm` 的实现结束了，接下来要做的就是向这个网络输入数据进行学习。在实现用于学习的代码之前，我们先来讨论一下语言模型的评价方法。

语言模型基于给定的已经出现的单词（信息）输出将要出现的单词的概率分布。困惑度（perplexity）常被用作评价语言模型的预测性能的指标。

简单地说，困惑度表示“概率的倒数”（这个解释在数据量为 1 时严格一致）。为了说明概率的倒数，我们仍旧考虑“you say goodbye and i say hello.”这一语料库。假设在向语言模型“模型 1”传入单词 you 时会输出图 5-32 的左图所示的概率分布。此时，下一个出现的单词是 say 的概率为 0.8，这是一个相当不错的预测。取这个概率的倒数，可以计算出困惑度为 $\frac{1}{0.8}=1.25$。

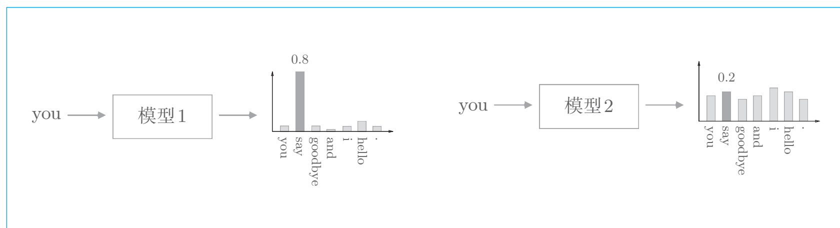


图 5-32　当输入单词 you 时，模型输出下一个出现的单词的概率分布

而图 5-32 右侧的模型（“模型 2”）预测出的正确单词的概率为 0.2，这显然是一个很差的预测，此时的困惑度为 $\frac{1}{0.2}=5$。

总结一下，“模型 1”能准确地预测，困惑度是 1.25；“模型 2”的预测未能命中，困惑度是 5.0。此例表明，困惑度越小越好。

那么，如何直观地解释值 1.25 和 5.0 呢？它们可以解释为“分叉度”。所谓分叉度，是指下一个可以选择的选项的数量（下一个可能出现的单词的候选个数）。在刚才的例子中，好的预测模型的分叉度是 1.25，这意味着下一个要出现的单词的候选个数可以控制在 1 个左右。而在差的模型中，下一个单词的候选个数有 5 个。



如上面的例子所示，基于困惑度可以评价模型的预测性能。好的模型可以高概率地预测出正确单词，所以困惑度较小（困惑度的最小值是 1.0）；而差的模型只能低概率地预测出正确单词，困惑度较大。

以上都是输入数据为 1 个时的困惑度。那么，在输入数据为多个的情况下，结果会怎样呢？我们可以根据下面的式子进行计算。

$$L = -\frac{1}{N}\sum_{n}\sum_{k} t_{nk}\log y_{nk} \tag{5.12}$$

$$\text{困惑度} = \mathrm{e}^{L} \tag{5.13}$$

这里，假设数据量为 N 个。$\boldsymbol{t}_n$ 是 one-hot 向量形式的正确解标签，t_{nk} 表示第 n 个数据的第 k 个值，y_{nk} 表示概率分布（神经网络中的 Softmax 的输出）。顺便说一下，L 是神经网络的损失，和式 (1.8) 完全相同，使用这个 L 计算出的 e^L 就是困惑度。

式子 (5.12) 看上去有些复杂，但是前面我们介绍的数据量为 1 时的“概率的倒数”“分叉度”“候选个数”等在这里也通用。也就是说，困惑度越小，分叉度越小，表明模型越好。



在信息论领域，困惑度也称为“平均分叉度”。这可以解释为，数据量为 1 时的分叉度是数据量为 N 时的分叉度的平均值。

5.5.3 RNNLM 的学习代码

下面，我们使用 PTB 数据集进行学习，不过这里仅使用 PTB 数据集（训练数据）的前 1000 个单词。这是因为在本节实现的 RNNLM 中，即便使用所有的训练数据，也得不出好的结果。下一章我们将对它进行改进。下面我们先来看一下学习用的代码（✏ ch05/train_custom_loop.py）。

```
import sys
sys.path.append('..')
import matplotlib.pyplot as plt
import numpy as np
from common.optimizer import SGD
from dataset import ptb
from simple_rnnlm import SimpleRnnlm


# 设定超参数
batch_size = 10
wordvec_size = 100
hidden_size = 100 # RNN的隐藏状态向量的元素个数
time_size = 5 # Truncated BPTT的时间跨度大小
lr = 0.1
max_epoch = 100

# 读入训练数据(缩小了数据集)
corpus, word_to_id, id_to_word = ptb.load_data('train')
corpus_size = 1000
corpus = corpus[:corpus_size]
vocab_size = int(max(corpus) + 1)

xs = corpus[:-1] # 输入
ts = corpus[1:] # 输出(监督标签)
data_size = len(xs)
print('corpus size: %d, vocabulary size: %d' % (corpus_size, vocab_size))

# 学习用的参数
max_iters = data_size // (batch_size * time_size)
time_idx = 0
total_loss = 0
```

```
loss_count = 0
ppl_list = []

# 生成模型
model = SimpleRnnlm(vocab_size, wordvec_size, hidden_size)
optimizer = SGD(lr)

# ❶ 计算读入mini-batch的各笔样本数据的开始位置
jump = (corpus_size - 1) // batch_size
offsets = [i * jump for i in range(batch_size)]

for epoch in range(max_epoch):
    for iter in range(max_iters):
        # ❷ 获取mini-batch
        batch_x = np.empty((batch_size, time_size), dtype='i')
        batch_t = np.empty((batch_size, time_size), dtype='i')
        for t in range(time_size):
            for i, offset in enumerate(offsets):
                batch_x[i, t] = xs[(offset + time_idx) % data_size]
                batch_t[i, t] = ts[(offset + time_idx) % data_size]
            time_idx += 1

        # 计算梯度，更新参数
        loss = model.forward(batch_x, batch_t)
        model.backward()
        optimizer.update(model.params, model.grads)
        total_loss += loss
        loss_count += 1

    # ❸ 各个epoch的困惑度评价
    ppl = np.exp(total_loss / loss_count)
    print('| epoch %d | perplexity %.2f'
          % (epoch+1, ppl))
    ppl_list.append(float(ppl))
    total_loss, loss_count = 0, 0
```

以上就是学习用的代码，这和我们之前看到的神经网络的学习基本上是一样的。不过，从宏观上看，仍有两点和之前的学习代码不同，即“数据的输入方式”和“困惑度的计算”。这里，我们将重点关注这两点，并对代码进行说明。

首先是数据的输入方式。这里我们使用 Truncated BPTT 进行学习，因此数据需要按顺序输入，并且 mini-batch 的各批次要平移读入数据的开始位置。在源代码❶处，计算各批次读入数据的开始位置 `offsets`。`offsets` 的各个元素中存放了读入数据的开始位置（偏移量）。

接着，在源代码❷处，按顺序读入数据。首先准备容器 `batch_x` 和 `batch_t`，然后依次增加 `time_idx` 变量，将 `time_idx` 处的数据从语料库中取出。这里利用❶中计算好的 `offsets`，各批次增加偏移量。另外，当读入语料库的位置超过语料库大小时，为了回到语料库的开头处，将当前位置除以语料库大小后的余数作为索引使用。

最后，基于式 (5.12) 计算困惑度，这在代码❸处完成。为了求每个 epoch 的困惑度，需要计算每个 epoch 的平均损失，然后再据此求困惑度。

以上就是对代码的说明，现在我们看一下学习结果。在上面的代码中，各个 epoch 的困惑度的结果都保存在了 `perplexity_list` 中，我们可以将它绘制出来，如图 5-33 所示。

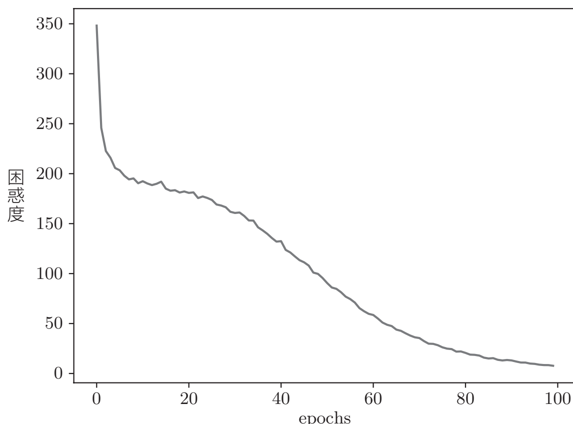


图 5-33　困惑度的演变

由图 5-33 可知，随着学习的进行，困惑度稳步下降。一开始超过 300 的困惑度到最后接近 1（最小值）了。不过这里使用的是很小的语料库，在实际情况下，当语料库增大时，现在的模型根本无法招架。下一章我们将指出当前 RNNLM 存在的问题，并进行改进。

5.5.4 RNNLM 的 Trainer 类

本书提供了用于学习 RNNLM 的 `RnnlmTrainer` 类，其内部封装了刚才的 RNNLM 的学习。将刚才的学习代码重构为 `RnnlmTrainer` 类，结果如下。这里只摘录源代码的一部分（全部代码在 `ch05/train.py` 中）。

```
...
from common.trainer import RnnlmTrainer

...
model = SimpleRnnlm(vocab_size, wordvec_size, hidden_size)
optimizer = SGD(lr)
trainer = RnnlmTrainer(model, optimizer)

trainer.fit(xs, ts, max_epoch, batch_size, time_size)
```

如上所示，首先使用 `model` 和 `optimizer` 初始化 `RnnlmTrainer` 类，然后调用 `fit()`，完成学习。此时，`RnnlmTrainer` 类的内部将执行上一节进行的一系列操作，具体如下所示。

- 按顺序生成 mini-batch
- 调用模型的正向传播和反向传播
- 使用优化器更新权重
- 评价困惑度



`RnnlmTrainer` 类与 1.4.4 节中介绍的 `Trainer` 类有相同的 API。神经网络的常规学习使用 `Trainer` 类，而 RNNLM 的学习则使用 `RnnlmTrainer` 类。

使用 `RnnlmTrainer` 类，可以避免每次写重复的代码。本书的剩余部分都将使用 `RnnlmTrainer` 类学习 RNNLM。

5.6 小结

本章的主题是 RNN。RNN 通过数据的循环，从过去继承数据并传递到现在和未来。如此，RNN 层的内部获得了记忆隐藏状态的能力。本书中我们花了很多时间说明 RNN 层的结构，并实现了 RNN 层（和 Time RNN 层）。

本章还利用 RNN 创建了语言模型。语言模型给单词序列赋概率值。特别地，条件语言模型从已经出现的单词序列计算下一个将要出现的单词的概率。通过构成利用了 RNN 的神经网络，理论上无论多么长的时序数据，都可以将它的重要信息记录在 RNN 的隐藏状态中。但是，在实际问题中，这样一来，许多情况下学习将无法顺利进行。下一章我们将指出 RNN 存在的问题，并研究替代 RNN 的 LSTM 层或 GRU 层。这些层在处理时序数据方面非常重要，被广泛用于前沿研究。

本章所学的内容

- RNN 具有环路，因此可以在内部记忆隐藏状态
- 通过展开 RNN 的循环，可以将其解释为多个 RNN 层连接起来的神经网络，可以通过常规的误差反向传播法进行学习（= BPTT）
- 在学习长时序数据时，要生成长度适中的数据块，进行以块为单位的 BPTT 学习（= Truncated BPTT）
- Truncated BPTT 只截断反向传播的连接
- 在 Truncated BPTT 中，为了维持正向传播的连接，需要按顺序输入数据
- 语言模型将单词序列解释为概率
- 理论上，使用 RNN 层的条件语言模型可以记忆所有已出现单词的信息

第6章
Gated RNN

卸下包袱，轻装上阵。

——尼采

上一章的 RNN 存在环路，可以记忆过去的信息，其结构非常简单，易于实现。不过，遗憾的是，这个 RNN 的效果并不好。原因在于，许多情况下它都无法很好地学习到时序数据的长期依赖关系。

现在，上一章的简单 RNN 经常被名为 LSTM 或 GRU 的层所代替。实际上，当我们说 RNN 时，更多的是指 LSTM 层，而不是上一章的 RNN。顺便说一句，当需要明确指上一章的 RNN 时，我们会说“简单 RNN”或“Elman”。

LSTM 和 GRU 中增加了一种名为“门”的结构。基于这个门，可以学习到时序数据的长期依赖关系。本章我们将指出上一章的 RNN 的问题，介绍代替它的 LSTM 和 GRU 等“Gated RNN”。特别是我们将花很多时间研究 LSTM 的结构，并揭示它实现“长期记忆”的机制。此外，我们将使用 LSTM 创建语言模型，并展示它可以在实际数据上很好地学习。

6.1 RNN的问题

上一章介绍的 RNN 之所以不擅长学习时序数据的长期依赖关系，是因为 BPTT 会发生梯度消失和梯度爆炸的问题。本节我们将首先回顾一下上一章介绍的 RNN 层，并通过一个实际的例子来说明为什么 RNN 层不擅长长期记忆。

6.1.1 RNN的复习

RNN 层存在环路。如果展开它的循环，它将变成一个在水平方向上延伸的网络，如图 6-1 所示。

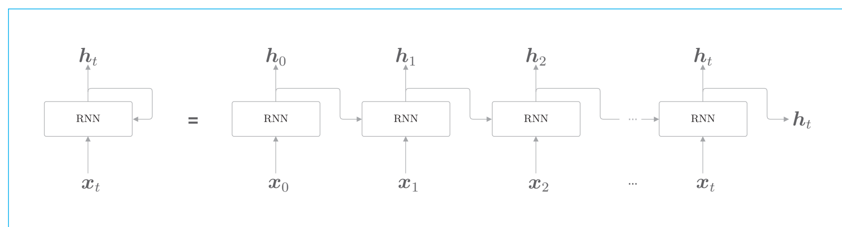


图6-1 RNN层：循环展开前和展开后

在图 6-1 中，当输入时序数据 $\boldsymbol{x}_t$ 时，RNN 层输出 $\boldsymbol{h}_t$。这个 $\boldsymbol{h}_t$ 也称为 RNN 层的隐藏状态，它记录过去的信息。

RNN 的特点在于使用了上一时刻的隐藏状态，由此，RNN 可以继承过去的信息。顺便说一下，如果用计算图来表示此时 RNN 层进行的处理，则有图 6-2。

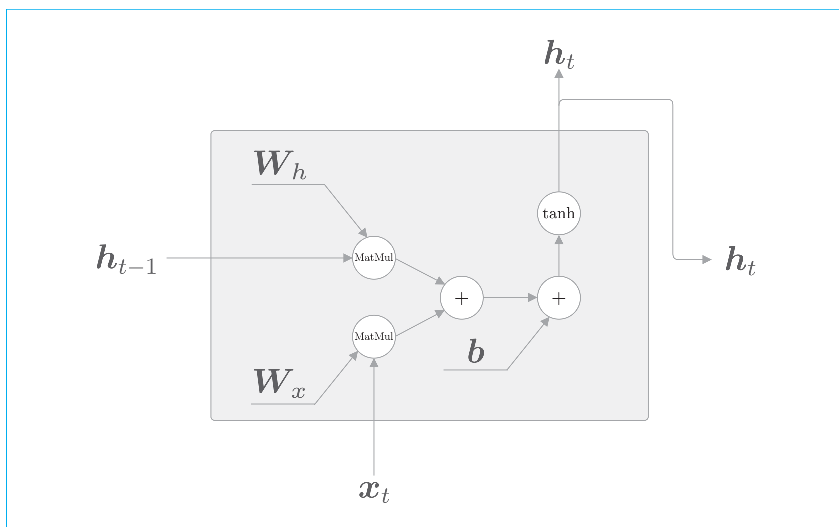


图 6-2　RNN 层的计算图（MatMul 节点表示矩阵乘积）

如图 6-2 所示，RNN 层的正向传播进行的计算由矩阵乘积、矩阵加法和基于激活函数 tanh 的变换构成，这就是我们上一章看到的 RNN 层。下面，我们看一下这个 RNN 层存在的问题（关于长期记忆的问题）。

6.1.2　梯度消失和梯度爆炸

语言模型的任务是根据已经出现的单词预测下一个将要出现的单词。上一章我们实现了基于 RNN 的语言模型 RNNLM，这里借着探讨 RNNLM 问题的机会，我们再来考虑一下图 6-3 所示的任务。

Tom was watching TV in his room. Mary came into the room. Mary said hi to [?]

图 6-3　某种程度上需要“长期记忆”的问题示例：“？”中应填入什么单词？

如前所述，填入“?”中的单词应该是 Tom。要正确回答这个问题，RNNLM 需要记住“Tom 在房间看电视，Mary 进了房间”这些信息。这些信息必须被编码并保存在 RNN 层的隐藏状态中。

现在让我们站在 RNNLM 进行学习的角度来考虑上述问题。在正确解标签为 Tom 时，RNNLM 中的梯度是如何传播的呢？这里我们使用 BPTT 进行学习，因此梯度将从正确解标签 Tom 出现的地方向过去的方向传播，如图 6-4 所示。

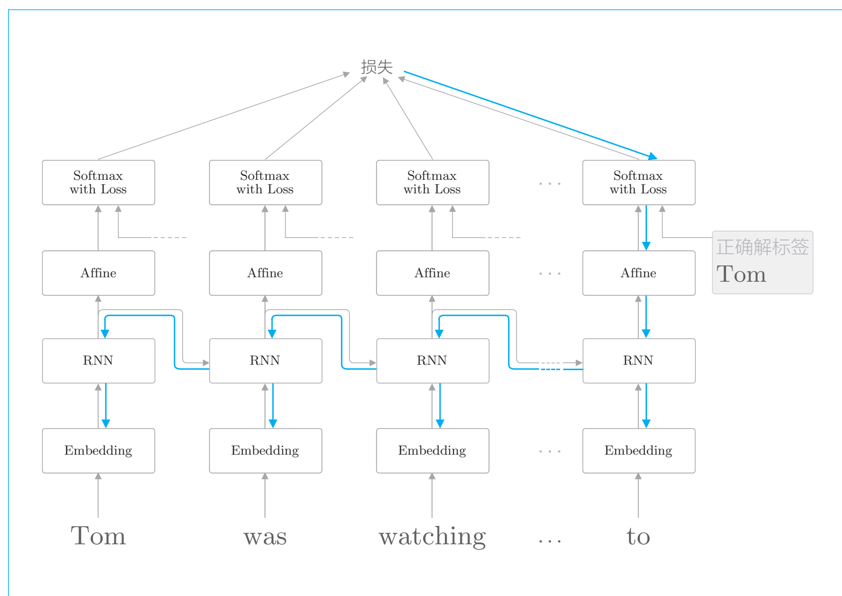


图 6-4 正确解标签为 Tom 时梯度的流动

在学习正确解标签 Tom 时，重要的是 RNN 层的存在。RNN 层通过向过去传递“有意义的梯度”，能够学习时间方向上的依赖关系。此时梯度（理论上）包含了那些应该学到的有意义的信息，通过将这些信息向过去传递，RNN 层学习长期的依赖关系。但是，如果这个梯度在中途变弱（甚至没有包含任何信息），则权重参数将不会被更新。也就是说，RNN 层无法学习长期的依赖关系。不幸的是，随着时间的回溯，这个简单 RNN 未能避免梯度变小（梯度消失）或者梯度变大（梯度爆炸）的命运。

6.1.3 梯度消失和梯度爆炸的原因

现在，我们深挖一下 RNN 层中梯度消失（或者梯度爆炸）的起因。如图 6-5 所示，这里仅关注 RNN 层在时间方向上的梯度传播。

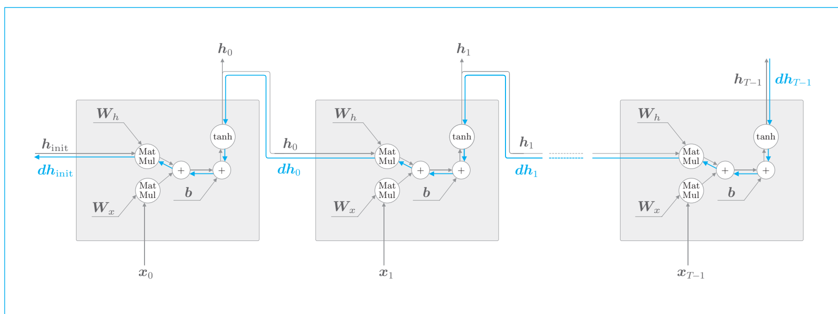


图 6-5　RNN 层在时间方向上的梯度传播

如图 6-5 所示， 这里考虑长度为 T 的时序数据，关注从第 T 个正确解标签传递出的梯度如何变化。就上面的问题来说，这相当于第 T 个正确解标签是 Tom 的情形。此时，关注时间方向上的梯度，可知反向传播的梯度流经 tanh、"+" 和 MatMul（矩阵乘积）运算。

"+" 的反向传播将上游传来的梯度原样传给下游，因此梯度的值不变。那么，剩下的 tanh 和 MatMul 运算会怎样变化呢？我们先来看一下 tanh。

附录 A 中会详细说明。当 $y = \tanh(x)$ 时，它的导数是 $\frac{\mathrm{d}y}{\mathrm{d}x} = 1 - y^2$。此时，将 $y = \tanh(x)$ 的值及其导数的值分别画在图上，如图 6-6 所示。

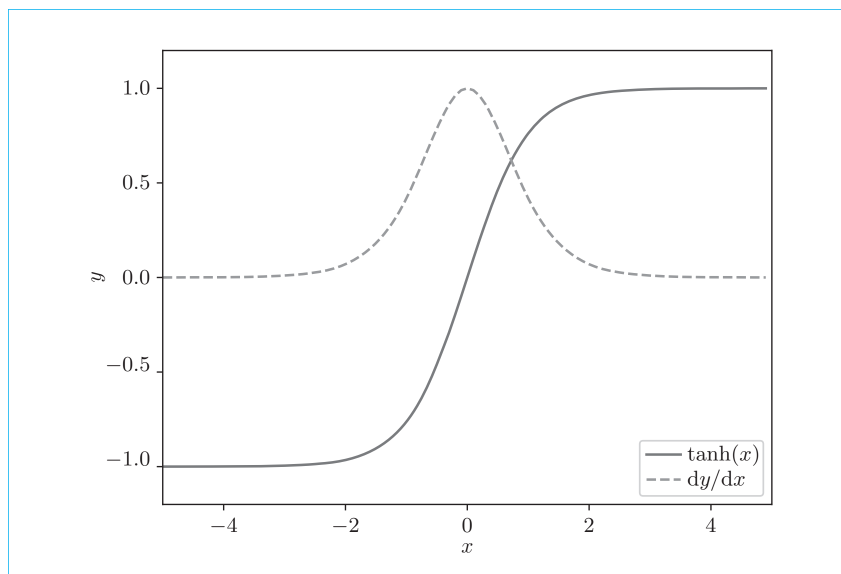


图6-6　$y = \tanh(x)$ 的图（虚线是导数）

图 6-6 中的虚线是 $y = \tanh(x)$ 的导数。从图中可以看出，它的值小于 1.0，并且随着 x 远离 0，它的值在变小。这意味着，当反向传播的梯度经过 tanh 节点时，它的值会越来越小。因此，如果经过 tanh 函数 T 次，则梯度也会减小 T 次。



RNN 层的激活函数一般使用 tanh 函数，但是如果改为 ReLU 函数，则有希望抑制梯度消失的问题（当 ReLU 的输入为 x 时，它的输出是 $\max(0, x)$）。这是因为，在 ReLU 的情况下，当 x 大于 0 时，反向传播将上游的梯度原样传递到下游，梯度不会"退化"。实际上，题为"Improving performance of recurrent neural network with relu nonlinearity"的论文 [29] 就使用 ReLU 实现了性能改善。

接下来，我们关注图 6-5 中的 MatMul（矩阵乘积）节点。简单起见，这里我们忽略图 6-5 中的 tanh 节点。如此一来，如图 6-7 所示，RNN 层的反向传播的梯度就仅取决于 MatMul 运算。

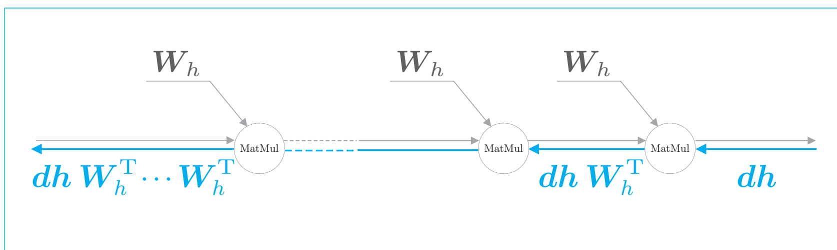


图 6-7 仅关注 RNN 层的矩阵乘积时的反向传播的梯度

在图 6-7 中，假定从上游传来梯度 $\boldsymbol{dh}$，此时 MatMul 节点的反向传播通过矩阵乘积 $\boldsymbol{dh}\boldsymbol{W}_h^{\mathrm{T}}$ 计算梯度。之后，根据时序数据的时间步长，将这个计算重复相应次数。这里需要注意的是，每一次矩阵乘积计算都使用相同的权重 $\boldsymbol{W}_h$。

那么，反向传播时梯度的值通过 MatMul 节点时会如何变化呢？一旦有了疑问，最好的方法就是做实验！让我们通过下面的代码，来观察梯度大小的变化（✏ ch06/rnn_gradient_graph.py）。

```
import numpy as np
import matplotlib.pyplot as plt

N = 2   # mini-batch的大小
H = 3   # 隐藏状态向量的维数
T = 20   # 时序数据的长度

dh = np.ones((N, H))
np.random.seed(3)  # 为了复现，固定随机数种子
Wh = np.random.randn(H, H)

norm_list = []
for t in range(T):
    dh = np.dot(dh, Wh.T)
    norm = np.sqrt(np.sum(dh**2)) / N
    norm_list.append(norm)
```

这里用 np.ones() 初始化 dh（np.ones() 是所有元素均为 1 的矩阵）。然后，根据反向传播的 MatMul 节点的数量更新 dh 相应次数，并将各步的 dh

的大小（范数）添加到 `norm_list` 中。这里，`dh` 的大小是 mini-batch（`N` 笔）中的平均“L2 范数”。L2 范数对所有元素的平方和求平方根。

下面，我们将上述代码的执行结果（`norm_list`）画在图上，如图 6-8 所示。

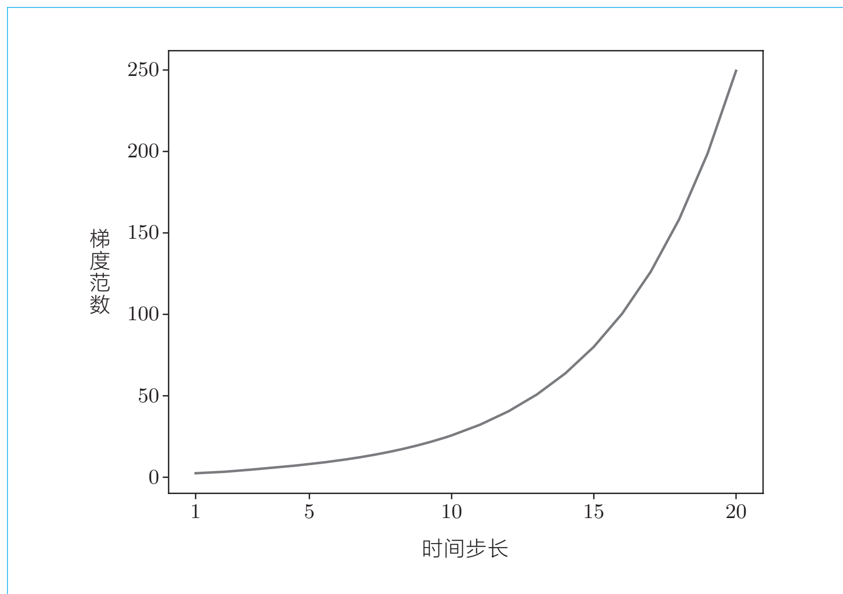


图 6-8　梯度 dh 的大小随时间步长呈指数级增加

如图 6-8 所示，可知梯度的大小随时间步长呈指数级增加，这就是梯度爆炸（exploding gradients）。如果发生梯度爆炸，最终就会导致溢出，出现 `NaN`（Not a Number，非数值）之类的值。如此一来，神经网络的学习将无法正确运行。

现在做第 2 个实验，将 `Wh` 的初始值改为下面的值。

```
# Wh = np.random.randn(H, H)       # before
Wh = np.random.randn(H, H) * 0.5   # after
```

使用这个初始值，进行与上面相同的实验，结果如图 6-9 所示。

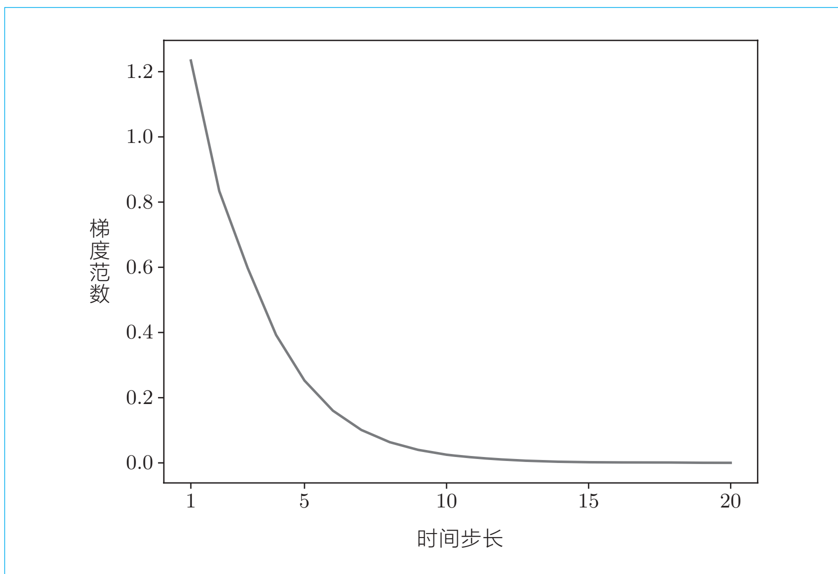


图 6-9 梯度 dh 的大小随时间步长呈指数级减小

从图 6-9 中可以看出，这次梯度呈指数级减小，这就是梯度消失（vanishing gradients）。如果发生梯度消失，梯度将迅速变小。一旦梯度变小，权重梯度不能被更新，模型就会无法学习长期的依赖关系。

在这里进行的实验中，梯度的大小或者呈指数级增加，或者呈指数级减小。为什么会出现这样的指数级变化呢？因为矩阵 Wh 被反复乘了 T 次。如果 Wh 是标量，则问题将很简单：当 Wh 大于 1 时，梯度呈指数级增加；当 Wh 小于 1 时，梯度呈指数级减小。

那么，如果 Wh 不是标量，而是矩阵呢？此时，矩阵的奇异值将成为指标。简单而言，矩阵的奇异值表示数据的离散程度。根据这个奇异值（更准确地说是多个奇异值中的最大值）是否大于 1，可以预测梯度大小的变化。



如果奇异值的最大值大于1，则可以预测梯度很有可能会呈指数级增加；而如果奇异值的最大值小于1，则可以判断梯度会呈指数级减小。但是，并不是说奇异值比1大就一定会出现梯度爆炸。也就是说，这是必要条件，并非充分条件。文献 [30] 中详细探讨了RNN的梯度消失和梯度爆炸问题，感兴趣的读者可以参考一下。

6.1.4 梯度爆炸的对策

至此，我们探讨了 RNN 的梯度爆炸和梯度消失问题，现在我们继续讨论解决方案。首先来看一下梯度爆炸。

解决梯度爆炸有既定的方法，称为梯度裁剪（gradients clipping）。这是一个非常简单的方法，它的伪代码如下所示：

$$\text{if} \quad \|\hat{\boldsymbol{g}}\| \geqslant threshold:$$

$$\hat{\boldsymbol{g}} = \frac{threshold}{\|\hat{\boldsymbol{g}}\|}\hat{\boldsymbol{g}}$$

这里假设可以将神经网络用到的所有参数的梯度整合成一个，并用符号 $\hat{\boldsymbol{g}}$ 表示。另外，将阈值设置为 $threshold$。此时，如果梯度的 L2 范数 $\|\hat{\boldsymbol{g}}\|$ 大于或等于阈值，就按上述方法修正梯度，这就是梯度裁剪。如你所见，虽然这个方法很简单，但是在许多情况下效果都不错。



$\hat{\boldsymbol{g}}$ 整合了神经网络中用到的所有参数的梯度。比如，当某个模型有 W1 和 W2 两个参数时，$\hat{\boldsymbol{g}}$ 就是这两个参数对应的梯度 dW1 和 dW2 的组合。

现在，我们用 Python 来实现梯度裁剪，将其实现为 clip_grads(grads, max_norm) 函数。参数 grads 是梯度的列表，max_norm 是阈值，此时梯度裁剪可以如下实现（✏ ch06/clip_grads.py）。

```
import numpy as np


dW1 = np.random.rand(3, 3) * 10
dW2 = np.random.rand(3, 3) * 10
grads = [dW1, dW2]
max_norm = 5.0

def clip_grads(grads, max_norm):
    total_norm = 0
    for grad in grads:
        total_norm += np.sum(grad ** 2)
    total_norm = np.sqrt(total_norm)

    rate = max_norm / (total_norm + 1e-6)
    if rate < 1:
        for grad in grads:
            grad *= rate


clip_grads(grads, max_norm)
```

这就是梯度裁剪的实现，并没有什么特别难的地方。因为将来还会用到 clip_grads(grads, max_norm)，所以我们在 common/util.py 中也放了一份相同的代码。



本书提供了用于RNNLM学习的RnnlmTrainer类（common/trainer.py），它的内部利用了上述梯度裁剪以防止梯度爆炸。我们会在6.4节再次说明RnnlmTrainer类中的梯度裁剪。

以上就是对梯度裁剪的说明。下面，我们看一下防止梯度消失的对策。

6.2　梯度消失和LSTM

在 RNN 的学习中，梯度消失也是一个大问题。为了解决这个问题，需要从根本上改变 RNN 层的结构，这里本章的主题 Gated RNN 就要登场了。

人们已经提出了诸多 Gated RNN 框架（网络结构），其中具有代表性的有 LSTM 和 GRU。本节我们将关注 LSTM，仔细研究它的结构，并阐明为何它不会（难以）引起梯度消失。另外，附录 C 中会对 GRU 进行说明。

6.2.1　LSTM 的接口

接下来，我们仔细看一下 LSTM 层。在此之前，为了将来方便，我们在计算图中引入“简略图示法”。如图 6-10 所示，这种图示法将矩阵计算等整理为一个长方形节点。

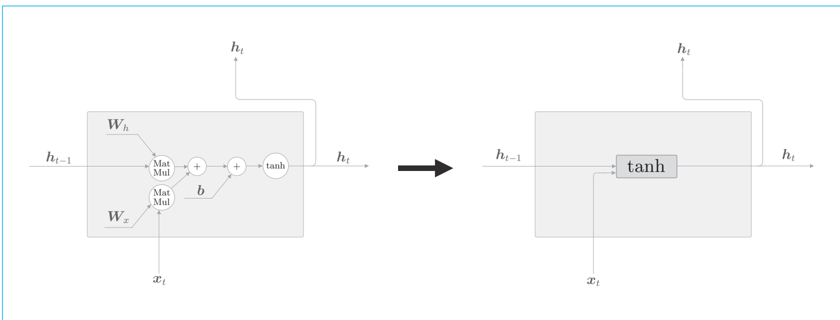


图 6-10　应用了简略图示法的 RNN 层：本节使用简略图示法以方便观察

如图 6-10 所示，这里将 $\tanh(\boldsymbol{h}_{t-1}\boldsymbol{W}_h + \boldsymbol{x}_t\boldsymbol{W}_x + \boldsymbol{b})$ 这个计算表示为一个长方形节点 tanh（$\boldsymbol{h}_{t-1}$ 和 $\boldsymbol{x}_t$ 是行向量），这个长方形节点中包含了矩阵乘积、偏置的和以及基于 tanh 函数的变换。

现在我们已经做好了准备。首先，我们来比较一下 LSTM 与 RNN 的接口（输入和输出）（图 6-11）。

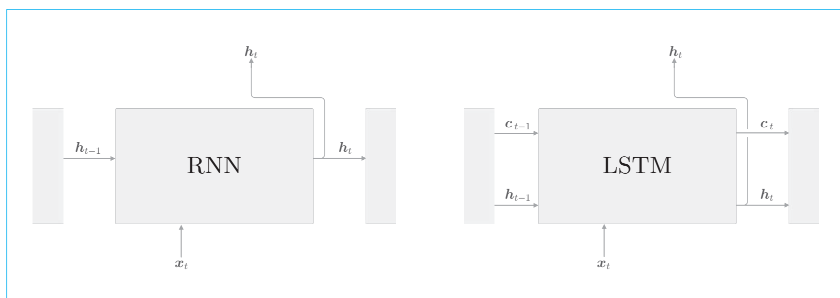


图6-11　RNN层与LSTM层的比较

如图 6-11 所示，LSTM 与 RNN 的接口的不同之处在于，LSTM 还有路径 $\boldsymbol{c}$。这个 $\boldsymbol{c}$ 称为记忆单元（或者简称为“单元”），相当于 LSTM 专用的记忆部门。

记忆单元的特点是，仅在 LSTM 层内部接收和传递数据。也就是说，记忆单元在 LSTM 层内部结束工作，不向其他层输出。而 LSTM 的隐藏状态 $\boldsymbol{h}$ 和 RNN 层相同，会被（向上）输出到其他层。



从接收LSTM的输出的一侧来看，LSTM的输出仅有隐藏状态向量 $\boldsymbol{h}$。记忆单元 $\boldsymbol{c}$ 对外部不可见，我们甚至不用考虑它的存在。

6.2.2　LSTM层的结构

现在，我们来看一下 LSTM 层的内部结构。这里，我想一个一个地组装 LSTM 的部件，并仔细研究它们的结构。以下内容参考了“colah's blog: Understanding LSTM Networks”[31] 这篇优秀的文章。

如前所述，LSTM 有记忆单元 $\boldsymbol{c}_t$。这个 $\boldsymbol{c}_t$ 存储了时刻 t 时 LSTM 的记忆，可以认为其中保存了从过去到时刻 t 的所有必要信息（或者以此为目的进行了学习）。然后，基于这个充满必要信息的记忆，向外部的层（和下一时刻的 LSTM）输出隐藏状态 $\boldsymbol{h}_t$。如图 6-12 所示，LSTM 输出经 tanh 函数变换后的记忆单元。

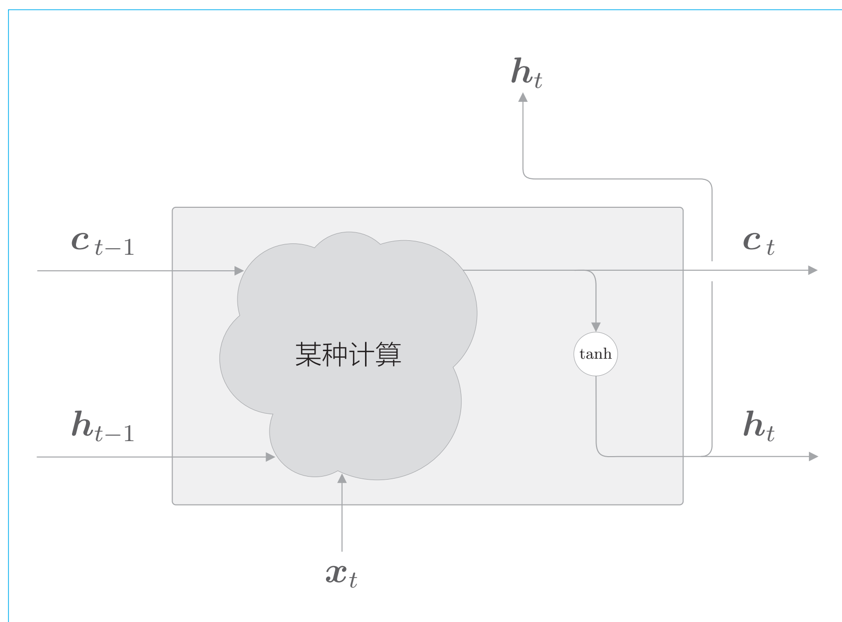


图6-12 LSTM层基于记忆单元 c_t 计算隐藏状态 h_t

如图 6-12 所示，当前的记忆单元 $\boldsymbol{c}_t$ 是基于 3 个输入 $\boldsymbol{c}_{t-1}$、$\boldsymbol{h}_{t-1}$ 和 $\boldsymbol{x}_t$，经过“某种计算”（后述）算出来的。这里的重点是隐藏状态 $\boldsymbol{h}_t$ 要使用更新后的 $\boldsymbol{c}_t$ 来计算。另外，这个计算是 $\boldsymbol{h}_t = \tanh(\boldsymbol{c}_t)$，表示对 $\boldsymbol{c}_t$ 的各个元素应用 tanh 函数。



到目前为止，记忆单元 $\boldsymbol{c}_t$ 和隐藏状态 $\boldsymbol{h}_t$ 的关系只是按元素应用 tanh 函数。这意味着，记忆单元 $\boldsymbol{c}_t$ 和隐藏状态 $\boldsymbol{h}_t$ 的元素个数相同。如果记忆单元 $\boldsymbol{c}_t$ 的元素个数是 100，则隐藏状态 $\boldsymbol{h}_t$ 的元素个数也是 100。

在进入下一项之前，我们先简单说明一下 Gate 的功能。Gate 是“门”的意思，就像将门打开或合上一样，控制数据的流动。直观上，如图 6-13 所示，门的作用就是阻止或者释放水流。



图6-13　门的比喻：控制水流

LSTM 中使用的门并非只能“开或合”，还可以根据将门打开多少来控制水的流量。如图 6-14 所示，可以将“开合程度”控制在 0.7（70%）或者 0.2（20%）。

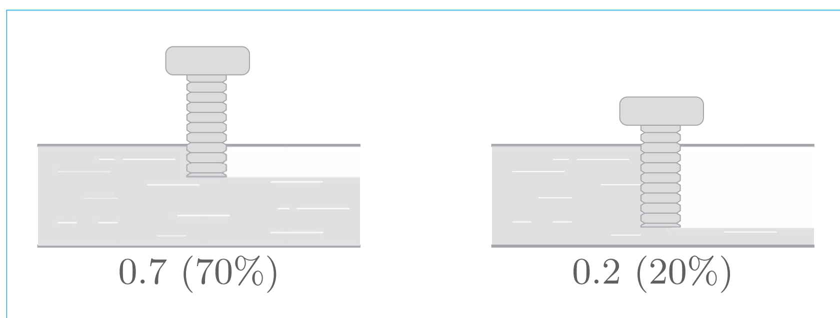


图6-14　将水的流量控制在0.0 ~ 1.0的范围内

如图 6-14 所示，门的开合程度由 0.0 ~1.0 的实数表示（1.0 为全开），通过这个数值控制流出的水量。这里的重点是，门的开合程度也是（自动）从数据中学习到的。



有专门的权重参数用于控制门的开合程度，这些权重参数通过学习被更新。另外，sigmoid 函数用于求门的开合程度（sigmoid 函数的输出范围在 0.0 ~ 1.0）。

6.2.3 输出门

现在，我们将话题转回到 LSTM。在刚才的说明中，隐藏状态 $\boldsymbol{h}_t$ 对记忆单元 $\boldsymbol{c}_t$ 仅仅应用了 tanh 函数。这里考虑对 $\tanh(\boldsymbol{c}_t)$ 施加门。换句话说，针对 $\tanh(\boldsymbol{c}_t)$ 的各个元素，调整它们作为下一时刻的隐藏状态的重要程度。由于这个门管理下一个隐藏状态 $\boldsymbol{h}_t$ 的输出，所以称为输出门（output gate）。

输出门的开合程度（流出比例）根据输入 $\boldsymbol{x}_t$ 和上一个状态 $\boldsymbol{h}_{t-1}$ 求出。此时进行的计算如式（6.1）所示。这里在使用的权重参数和偏置的上标上添加了 output 的首字母 o。之后，我们也将使用上标表示门。另外，sigmoid 函数用 $\sigma()$ 表示。

$$\boldsymbol{o} = \sigma(\boldsymbol{x}_t \boldsymbol{W}_x^{(\mathrm{o})} + \boldsymbol{h}_{t-1} \boldsymbol{W}_h^{(\mathrm{o})} + \boldsymbol{b}^{(\mathrm{o})}) \tag{6.1}$$

如式 (6.1) 所示，输入 $\boldsymbol{x}_t$ 有权重 $\boldsymbol{W}_x{}^{(\mathrm{o})}$，上一时刻的状态 $\boldsymbol{h}_{t-1}$ 有权重 $\boldsymbol{W}_h{}^{(\mathrm{o})}$（$\boldsymbol{x}_t$ 和 $\boldsymbol{h}_{t-1}$ 是行向量）。将它们的矩阵乘积和偏置 $\boldsymbol{b}^{(\mathrm{o})}$ 之和传给 sigmoid 函数，结果就是输出门的输出 $\boldsymbol{o}$。最后，将这个 $\boldsymbol{o}$ 和 $\tanh(\boldsymbol{c}_t)$ 的对应元素的乘积作为 $\boldsymbol{h}_t$ 输出。将这些计算绘制成计算图，结果如图 6-15 所示。

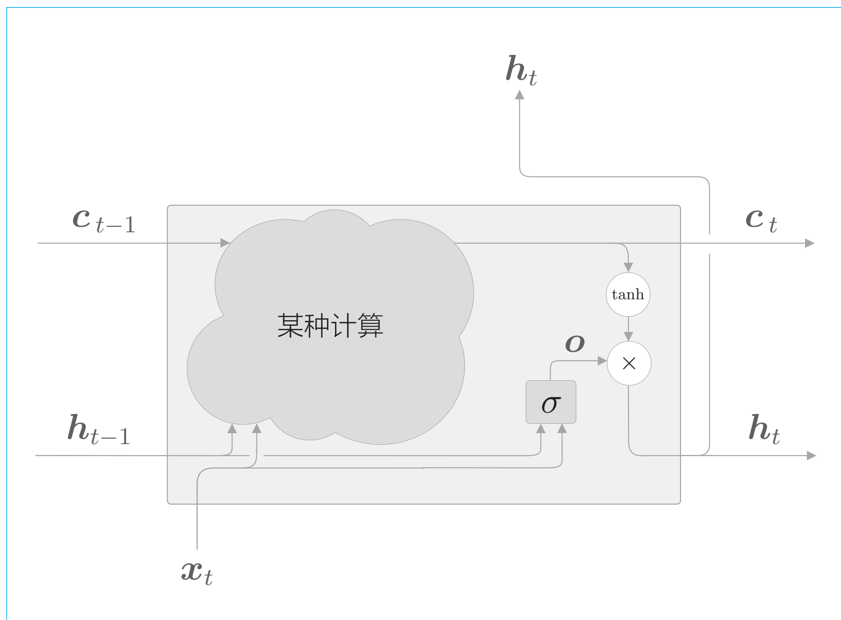


图6-15　添加输出门

在图 6-15 中，将输出门进行的式 (6.1) 的计算表示为 σ。然后，将它的输出表示为 $\boldsymbol{o}$，则 $\boldsymbol{h}_t$ 可由 $\boldsymbol{o}$ 和 $\tanh(\boldsymbol{c}_t)$ 的乘积计算出来。这里说的“乘积”是对应元素的乘积，也称为阿达玛乘积。如果用 $\odot$ 表示阿达玛乘积，则此处的计算如下所示：

$$\boldsymbol{h}_t = \boldsymbol{o} \odot \tanh(\boldsymbol{c}_t) \tag{6.2}$$

以上就是 LSTM 的输出门。这样一来，LSTM 的输出部分就完成了，接着我们再来看一下记忆单元的更新部分。



tanh的输出是 $-1.0 \sim 1.0$的实数。我们可以认为这个 $-1.0 \sim 1.0$的数值表示某种被编码的“信息”的强弱(程度)。而sigmoid函数的输出是0.0~1.0的实数，表示数据流出的比例。因此，在大多数情况下，门使用sigmoid函数作为激活函数，而包含实质信息的数据则使用tanh函数作为激活函数。

6.2.4 遗忘门

只有放下包袱，才能轻装上路。接下来，我们要做的就是明确告诉记忆单元需要“忘记什么”。这里，我们使用门来实现这一目标。

现在，我们在记忆单元 $\boldsymbol{c}_{t-1}$ 上添加一个忘记不必要记忆的门，这里称为遗忘门（forget gate）。将遗忘门添加到LSTM层，计算图如图6-16所示。

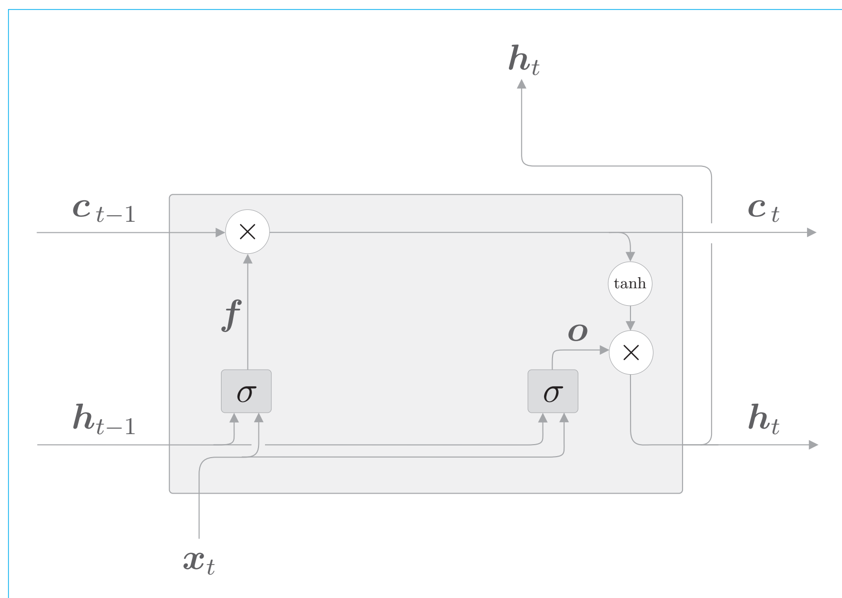


图6-16 添加遗忘门

在图6-16中，将遗忘门进行的一系列计算表示为 σ，其中有遗忘门专用的权重参数，此时的计算如下：

$$\boldsymbol{f} = \sigma(\boldsymbol{x}_t \boldsymbol{W}_x^{(\mathrm{f})} + \boldsymbol{h}_{t-1} \boldsymbol{W}_h^{(\mathrm{f})} + \boldsymbol{b}^{(\mathrm{f})}) \tag{6.3}$$

遗忘门的输出 $\boldsymbol{f}$ 可以由式(6.3)求得。然后，$\boldsymbol{c}_t$ 由这个 $\boldsymbol{f}$ 和上一个记忆单元 $\boldsymbol{c}_{t-1}$ 的对应元素的乘积求得（$\boldsymbol{c}_t = \boldsymbol{f} \odot \boldsymbol{c}_{t-1}$）。

6.2.5 新的记忆单元

遗忘门从上一时刻的记忆单元中删除了应该忘记的东西，但是这样一来，记忆单元只会忘记信息。现在我们还想向这个记忆单元添加一些应当记住的新信息，为此我们添加新的 tanh 节点（图 6-17）。

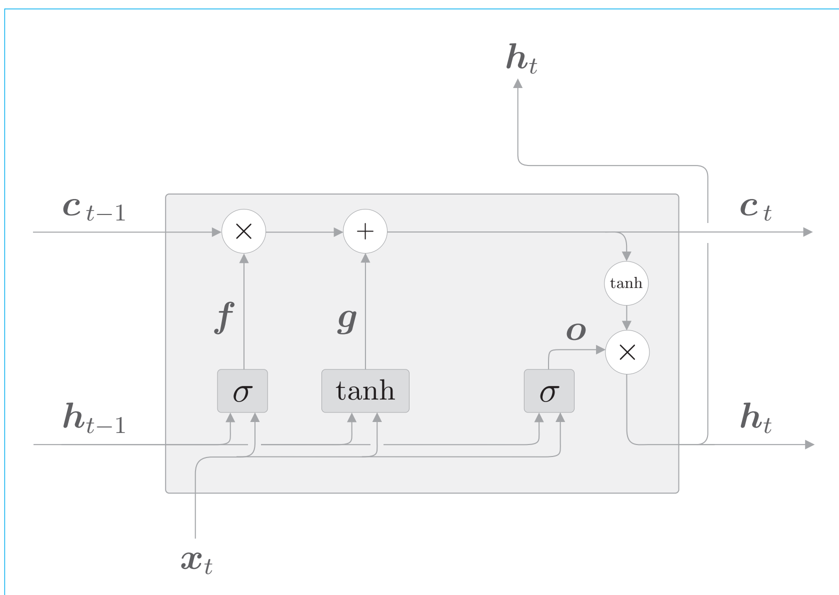


图6-17 向新的记忆单元添加必要信息

如图 6-17 所示，基于 tanh 节点计算出的结果被加到上一时刻的记忆单元 $\boldsymbol{c}_{t-1}$ 上。这样一来，新的信息就被添加到了记忆单元中。这个 tanh 节点的作用不是门，而是将新的信息添加到记忆单元中。因此，它不用 sigmoid 函数作为激活函数，而是使用 tanh 函数。tanh 节点进行的计算如下所示：

$$\boldsymbol{g} = \tanh(\boldsymbol{x}_t \boldsymbol{W}_x^{(\mathrm{g})} + \boldsymbol{h}_{t-1} \boldsymbol{W}_h^{(\mathrm{g})} + \boldsymbol{b}^{(\mathrm{g})}) \tag{6.4}$$

这里用 $\boldsymbol{g}$ 表示向记忆单元添加的新信息。通过将这个 $\boldsymbol{g}$ 加到上一时刻的 $\boldsymbol{c}_{t-1}$ 上，从而形成新的记忆。

6.2.6 输入门

最后，我们给图 6-17 的 $\boldsymbol{g}$ 添加门，这里将这个新添加的门称为输入门（input gate）。添加输入门后，计算图如图 6-18 所示。

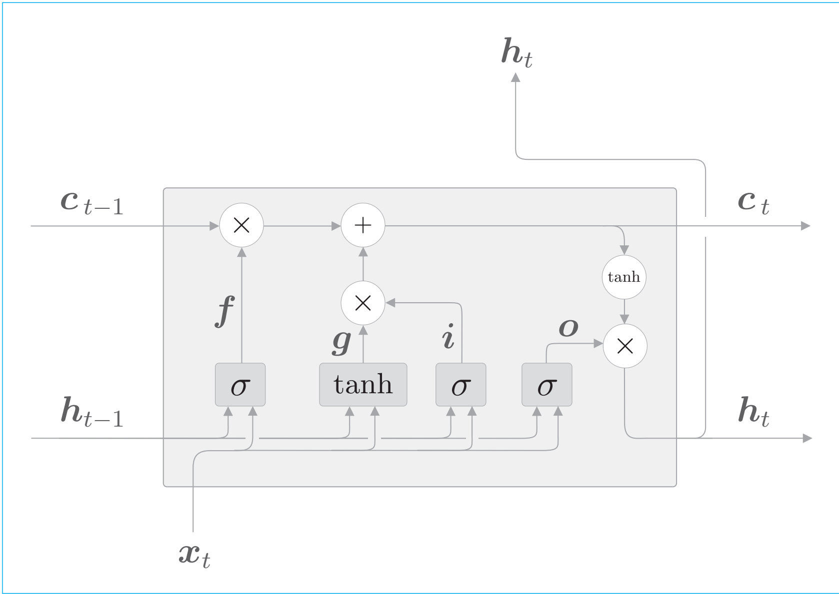


图 6-18 添加输入门

输入门判断新增信息 $\boldsymbol{g}$ 的各个元素的价值有多大。输入门不会不经考虑就添加新信息，而是会对要添加的信息进行取舍。换句话说，输入门会添加加权后的新信息。

在图 6-18 中，用 σ 表示输入门，用 $\boldsymbol{i}$ 表示输出，此时进行的计算如下所示：

$$\boldsymbol{i} = \sigma(\boldsymbol{x}_t \boldsymbol{W}_x^{(\mathrm{i})} + \boldsymbol{h}_{t-1} \boldsymbol{W}_h^{(\mathrm{i})} + \boldsymbol{b}^{(\mathrm{i})}) \tag{6.5}$$

然后，将 $\boldsymbol{i}$ 和 $\boldsymbol{g}$ 的对应元素的乘积添加到记忆单元中。以上就是对 LSTM 内部处理的说明。



LSTM有多个“变体”。这里说明的LSTM是最有代表性的LSTM，也有许多在门的连接方式上稍微不同的其他LSTM。

6.2.7 LSTM的梯度的流动

上面我们介绍了LSTM的结构，那么，为什么它不会引起梯度消失呢？其原因可以通过观察记忆单元 $\boldsymbol{c}$ 的反向传播来了解（图6-19）。

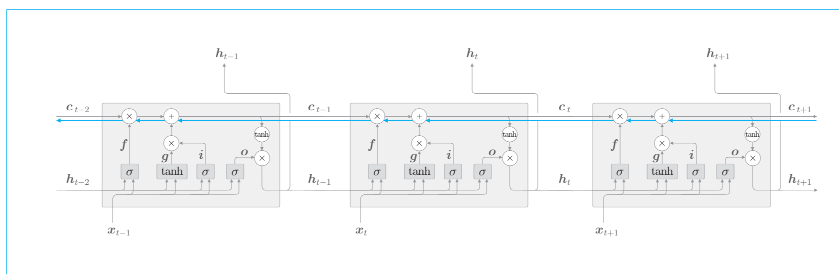


图6-19 记忆单元的反向传播

在图6-19中，我们仅关注记忆单元，绘制了它的反向传播。此时，记忆单元的反向传播仅流过“+”和“×”节点。“+”节点将上游传来的梯度原样流出，所以梯度没有变化（退化）。

而“×”节点的计算并不是矩阵乘积，而是对应元素的乘积（阿达玛积）。顺便说一下，在之前的RNN的反向传播中，我们使用相同的权重矩阵重复了多次矩阵乘积计算，由此导致了梯度消失（或梯度爆炸）。而这里的LSTM的反向传播进行的不是矩阵乘积计算，而是对应元素的乘积计算，而且每次都会基于不同的门值进行对应元素的乘积计算。这就是它不会发生梯度消失（或梯度爆炸）的原因。

图6-19的“×”节点的计算由遗忘门控制（每次输出不同的门值）。遗忘门认为“应该忘记”的记忆单元的元素，其梯度会变小；而遗忘门认为“不能忘记”的元素，其梯度在向过去的方向流动时不会退化。因此，可以期待记忆单元的梯度（应该长期记住的信息）能在不发生梯度消失的情况下传播。

从以上讨论可知，LSTM的记忆单元不会（难以）发生梯度消失。因此，可以期待记忆单元能够保存（学习）长期的依赖关系。



LSTM是Long Short-Term Memory（长短期记忆）的缩写，意思是可以长（Long）时间维持短期记忆（Short-Term Memory）。

6.3 LSTM的实现

下面，我们来实现LSTM。这里将进行单步处理的类实现为`LSTM`类，将整体处理`T`步的类实现为`TimeLSTM`类。现在我们先来整理一下LSTM中进行的计算，如下所示：

$$\begin{aligned}
\boldsymbol{f} &= \sigma(\boldsymbol{x}_t \boldsymbol{W}_x^{(\mathrm{f})} + \boldsymbol{h}_{t-1} \boldsymbol{W}_h^{(\mathrm{f})} + \boldsymbol{b}^{(\mathrm{f})}) \\
\boldsymbol{g} &= \tanh(\boldsymbol{x}_t \boldsymbol{W}_x^{(\mathrm{g})} + \boldsymbol{h}_{t-1} \boldsymbol{W}_h^{(\mathrm{g})} + \boldsymbol{b}^{(\mathrm{g})}) \\
\boldsymbol{i} &= \sigma(\boldsymbol{x}_t \boldsymbol{W}_x^{(\mathrm{i})} + \boldsymbol{h}_{t-1} \boldsymbol{W}_h^{(\mathrm{i})} + \boldsymbol{b}^{(\mathrm{i})}) \\
\boldsymbol{o} &= \sigma(\boldsymbol{x}_t \boldsymbol{W}_x^{(\mathrm{o})} + \boldsymbol{h}_{t-1} \boldsymbol{W}_h^{(\mathrm{o})} + \boldsymbol{b}^{(\mathrm{o})})
\end{aligned} \tag{6.6}$$

$$\boldsymbol{c}_t = \boldsymbol{f} \odot \boldsymbol{c}_{t-1} + \boldsymbol{g} \odot \boldsymbol{i} \tag{6.7}$$

$$\boldsymbol{h}_t = \boldsymbol{o} \odot \tanh(\boldsymbol{c}_t) \tag{6.8}$$

以上就是LSTM进行的计算。这里需要注意式(6.6)中的4个仿射变换。这里的仿射变换是指 $\boldsymbol{x}\boldsymbol{W}_x + \boldsymbol{h}\boldsymbol{W}_h + \boldsymbol{b}$ 这样的式子。式(6.6)中通过4个式子分别进行仿射变换，但其实可以整合为通过1个式子进行，如图6-20所示。

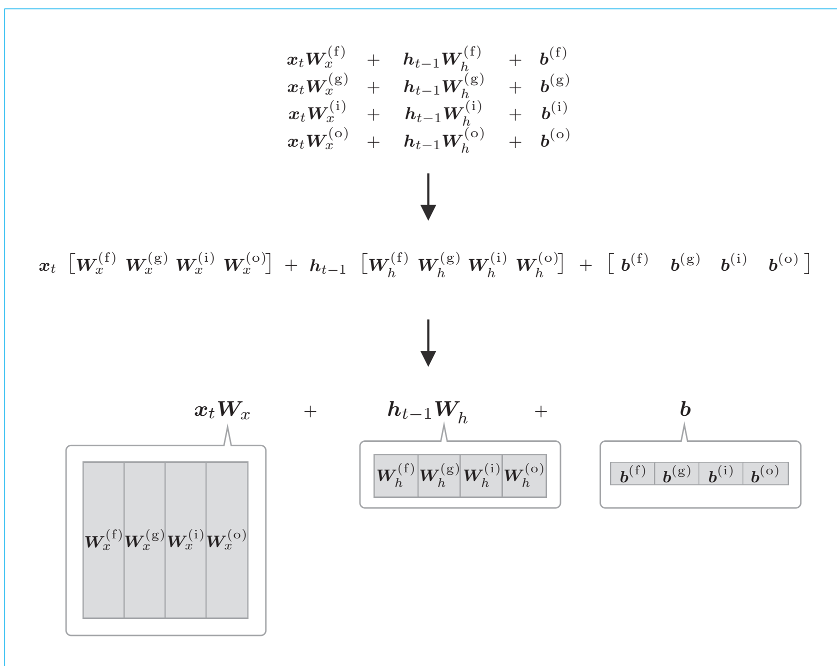


图6-20　整合4个权重，通过1次仿射变换进行4个计算

在图 6-20 中，4 个权重（或偏置）被整合为了 1 个。如此，原本单独执行 4 次的仿射变换通过 1 次计算即可完成，可以加快计算速度。这是因为矩阵库计算“大矩阵”时通常会更快，而且通过将权重整合到一起管理，源代码也会更简洁。

假设 $\boldsymbol{W}_x$、$\boldsymbol{W}_h$ 和 $\boldsymbol{b}$ 分别包含 4 个权重（或偏置），此时 LSTM 的计算图如图 6-21 所示。

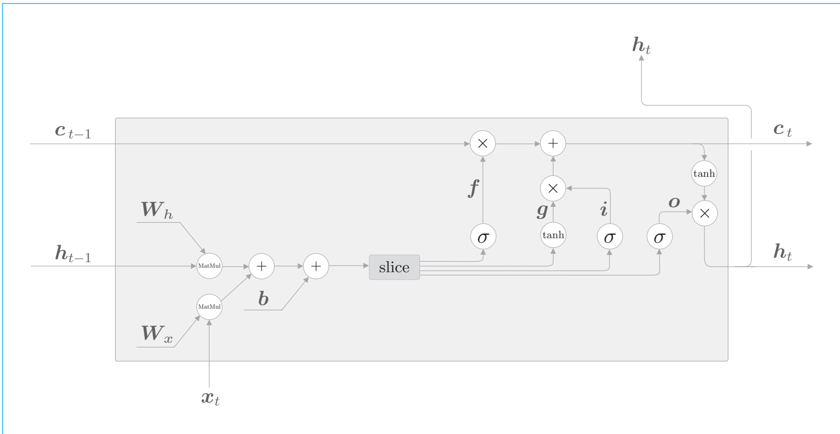


图6-21 整合4个权重进行仿射变换的LSTM的计算图

如图 6-21 所示，先一起执行 4 个仿射变换。然后，基于 slice 节点，取出 4 个结果。这个 slice 节点很简单，它将仿射变换的结果（矩阵）均等地分成 4 份，然后取出内容。在 slice 节点之后，数据流过激活函数（sigmoid 函数或 tanh 函数），进行上一节介绍的计算。

现在，参考图 6-21，我们来实现 LSTM 类。首先来看一下 LSTM 类的初始化代码（✏ common/time_layers.py）。

```
class LSTM:
    def __init__(self, Wx, Wh, b):
        self.params = [Wx, Wh, b]
        self.grads = [np.zeros_like(Wx), np.zeros_like(Wh),
↪   np.zeros_like(b)]
        self.cache = None
```

初始化的参数有权重参数 Wx、Wh 和偏置 b。如前所述，这些权重（或偏置）整合了 4 个权重。把这些参数获得的权重参数设定给成员变量 params，并初始化形状与之对应的梯度。另外，成员变量 cache 保存正向传播的中间结果，它们将在反向传播的计算中使用。

接下来实现正向传播的 forward(x, h_prev, c_prev) 方法。它的参数接收当前时刻的输入 x、上一时刻的隐藏状态 h_prev，以及上一时刻的记忆单

元 c_prev（✏ common/time_layers.py）。

```
def forward(self, x, h_prev, c_prev):
    Wx, Wh, b = self.params
    N, H = h_prev.shape

    A = np.dot(x, Wx) + np.dot(h_prev, Wh) + b

    # slice
    f = A[:, :H]
    g = A[:, H:2*H]
    i = A[:, 2*H:3*H]
    o = A[:, 3*H:]

    f = sigmoid(f)
    g = np.tanh(g)
    i = sigmoid(i)
    o = sigmoid(o)

    c_next = f * c_prev + g * i
    h_next = o * np.tanh(c_next)

    self.cache = (x, h_prev, c_prev, i, f, g, o, c_next)
    return h_next, c_next
```

首先进行仿射变换。重复一下，此时的成员变量 Wx、Wh 和 b 保存的是 4 个权重，矩阵的形状将变为如图 6-22 所示的样子。

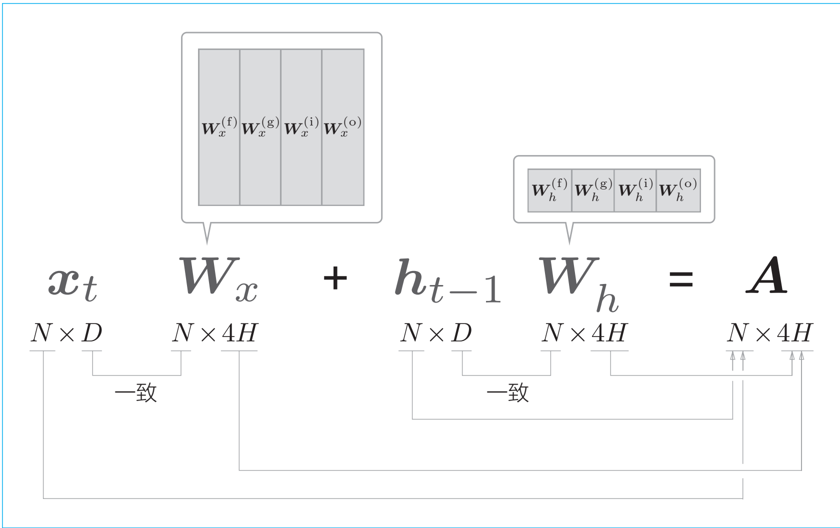


图6-22 仿射变换的形状的改变（省略偏置）

在图 6-22 中，批大小是 N，输入数据的维数是 D，记忆单元和隐藏状态的维数都是 H。另外，计算结果 A 中保存了 4 个仿射变换的结果。因此，通过 A[:, :H]、A[:, H:2*H] 这样的切片取出数据，并分配给之后的运算节点。参考 LSTM 的数学式和计算图，剩余的实现应该不难。



LSTM 层中保存了 4 个权重。这样一来，LSTM 层只需管理 Wx、Wh 和 b 这 3 个参数。顺便说一下，RNN 层中也保存着 Wx、Wh 和 b 这 3 个参数。LSTM 层和 RNN 层的参数数量虽然相同，但是它们的形状不一样。

LSTM 的反向传播可以通过将图 6-21 的计算图反方向传播而求得。基于前面介绍的知识，这并不困难。不过，因为 slide 节点是第一次见到，所以我们简要说明一下它的反向传播。

slice 节点将矩阵分成了 4 份，因此它的反向传播需要整合 4 个梯度，如图 6-23 所示。

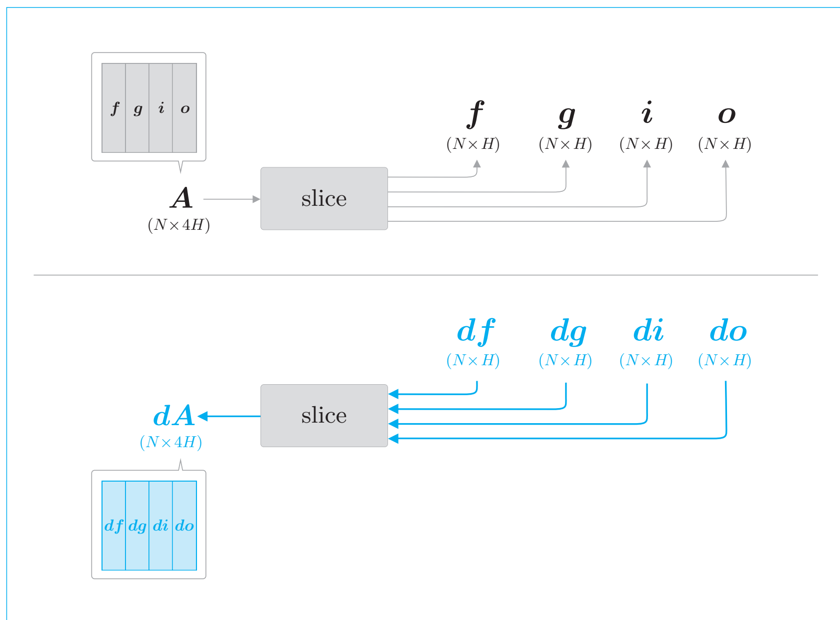


图6-23 slice节点的正向传播（上）和反向传播（下）

由图 6-23 可知，在 slice 节点的反向传播中，拼接 4 个矩阵。图中有 4 个梯度 `df`、`dg`、`di` 和 `do`，将它们拼接成 `dA`。如果通过 NumPy 进行，则可以使用 `np.hstack()`。`np.hstack()` 在水平方向上将参数中给定的数组拼接起来（垂直方向上的拼接使用 `np.vstack()`）。因此，上述处理可以用下面 1 行代码完成。

```
dA = np.hstack((df, dg, di, do))
```

以上就是对 slice 节点的反向传播的说明。

Time LSTM 层的实现

现在我们继续 `TimeLSTM` 的实现。Time LSTM 层是整体处理 T 个时序数据的层，由 T 个 LSTM 层构成，如图 6-24 所示。

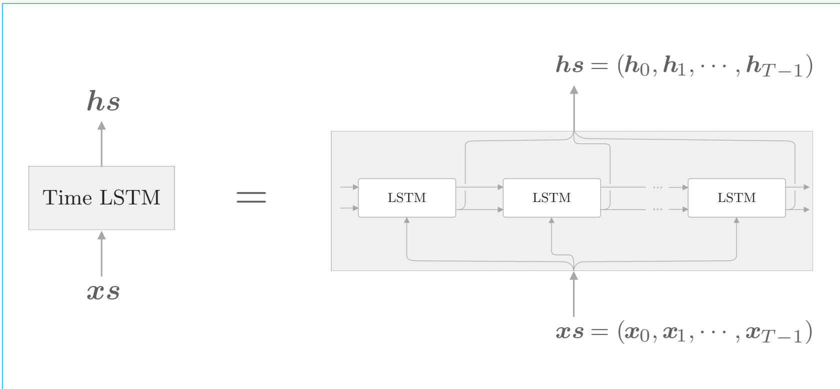


图6-24 Time LSTM的输入和输出

如前所述，RNN 中使用 Truncated BPTT 进行学习。Truncated BPTT 以适当的长度截断反向传播的连接，但是需要维持正向传播的数据流。为此，如图 6-25 所示，将隐藏状态和记忆单元保存在成员变量中。这样一来，在调用下一个 `forward()` 函数时，就可以继承上一时刻的隐藏状态（和记忆单元）。

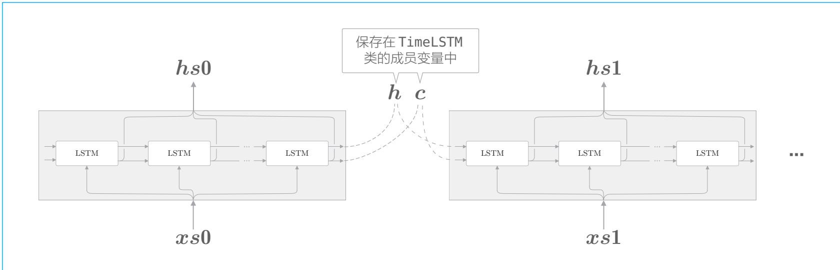


图6-25 Time LSTM的反向传播的输入和输出

我们已经实现了 Time RNN 层，这里也以同样的方式实现 Time LSTM 层。`TimeLSTM` 可以像下面这样实现（✏ `common/time_layers.py`）。

```
class TimeLSTM:
    def __init__(self, Wx, Wh, b, stateful=False):
        self.params = [Wx, Wh, b]
```

```
        self.grads = [np.zeros_like(Wx), np.zeros_like(Wh),
↪   np.zeros_like(b)]
        self.layers = None

        self.h, self.c = None, None
        self.dh = None
        self.stateful = stateful

    def forward(self, xs):
        Wx, Wh, b = self.params
        N, T, D = xs.shape
        H = Wh.shape[0]

        self.layers = []
        hs = np.empty((N, T, H), dtype='f')

        if not self.stateful or self.h is None:
            self.h = np.zeros((N, H), dtype='f')
        if not self.stateful or self.c is None:
            self.c = np.zeros((N, H), dtype='f')

        for t in range(T):
            layer = LSTM(*self.params)
            self.h, self.c = layer.forward(xs[:, t, :], self.h, self.c)
            hs[:, t, :] = self.h

            self.layers.append(layer)

        return hs

    def backward(self, dhs):
        Wx, Wh, b = self.params
        N, T, H = dhs.shape
        D = Wx.shape[0]

        dxs = np.empty((N, T, D), dtype='f')
        dh, dc = 0, 0

        grads = [0, 0, 0]
        for t in reversed(range(T)):
```

```
            layer = self.layers[t]
            dx, dh, dc = layer.backward(dhs[:, t, :] + dh, dc)
            dxs[:, t, :] = dx
            for i, grad in enumerate(layer.grads):
                grads[i] += grad

        for i, grad in enumerate(grads):
            self.grads[i][...] = grad
            self.dh = dh
            return dxs

    def set_state(self, h, c=None):
        self.h, self.c = h, c

    def reset_state(self):
        self.h, self.c = None, None
```

在 LSTM 中，除了隐藏状态 h 外，还使用记忆单元 c。TimeLSTM 类的实现和 TimeRNN 类几乎一样。这里仍通过参数 stateful 指定是否维持状态。接下来，我们使用这个 TimeLSTM 创建语言模型。

6.4 使用LSTM的语言模型

Time LSTM 层的实现完成了，现在我们来实现正题——语言模型。这里实现的语言模型和上一章几乎是一样的，唯一的区别是，上一章使用 Time RNN 层的地方这次使用 Time LSTM 层，如图 6-26 所示。

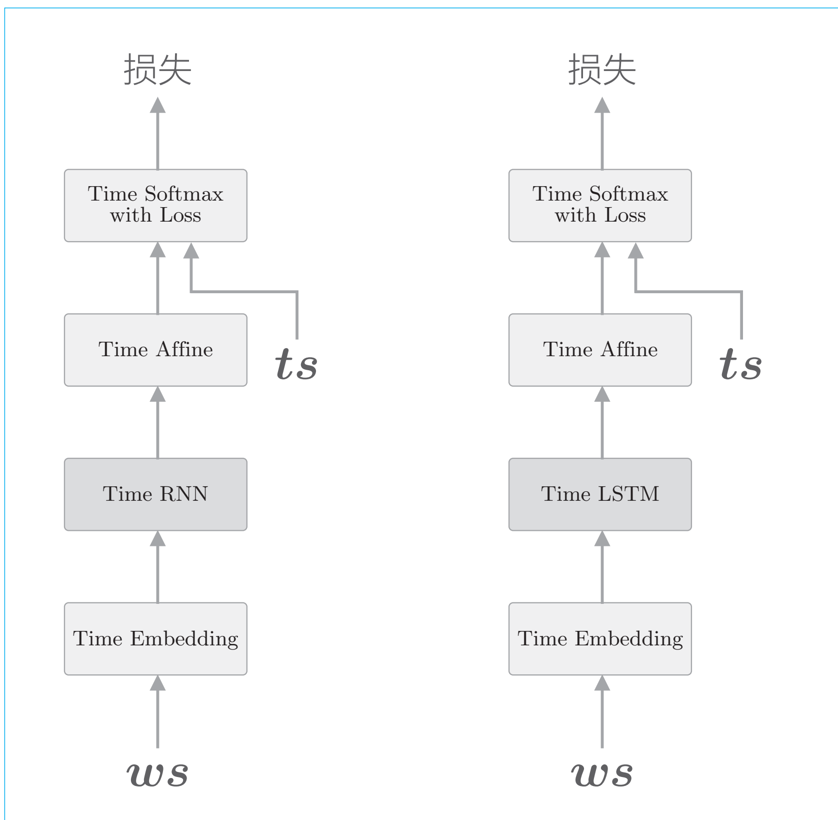


图6-26　语言模型的网络结构。左图是上一章创建的使用Time RNN的模型，右图是本章创建的使用Time LSTM的模型

由图6-26可知，这里和上一章实现的语言模型的差别在于使用了LSTM。我们将图6-26右图中的神经网络实现为Rnnlm类。Rnnlm类和上一章介绍的SimpleRnnlm类几乎相同，但是增加了一些新方法。下面给出使用LSTM层实现的Rnnlm类的代码[①]。

```
import sys
sys.path.append('..')
from common.time_layers import *
```

① 这里给出的代码对应于ch06/rnnlm.py。ch06/rnnlm.py通过继承BaseModel类，实现更为简略。

```
import pickle

class Rnnlm:
    def __init__(self, vocab_size=10000, wordvec_size=100,
 ↪  hidden_size=100):
        V, D, H = vocab_size, wordvec_size, hidden_size
        rn = np.random.randn

        # 初始化权重
        embed_W = (rn(V, D) / 100).astype('f')
        lstm_Wx = (rn(D, 4 * H) / np.sqrt(D)).astype('f')
        lstm_Wh = (rn(H, 4 * H) / np.sqrt(H)).astype('f')
        lstm_b = np.zeros(4 * H).astype('f')
        affine_W = (rn(H, V) / np.sqrt(H)).astype('f')
        affine_b = np.zeros(V).astype('f')

        # 生成层
        self.layers = [
            TimeEmbedding(embed_W),
            TimeLSTM(lstm_Wx, lstm_Wh, lstm_b, stateful=True),
            TimeAffine(affine_W, affine_b)
        ]
        self.loss_layer = TimeSoftmaxWithLoss()
        self.lstm_layer = self.layers[1]

        # 将所有的权重和梯度整理到列表中
        self.params, self.grads = [], []
        for layer in self.layers:
            self.params += layer.params
            self.grads += layer.grads

    def predict(self, xs):
        for layer in self.layers:
            xs = layer.forward(xs)
        return xs

    def forward(self, xs, ts):
        score = self.predict(xs)
        loss = self.loss_layer.forward(score, ts)
        return loss
```

```
    def backward(self, dout=1):
        dout = self.loss_layer.backward(dout)
        for layer in reversed(self.layers):
            dout = layer.backward(dout)
        return dout

    def reset_state(self):
        self.lstm_layer.reset_state()

    def save_params(self, file_name='Rnnlm.pkl'):
        with open(file_name, 'wb') as f:
            pickle.dump(self.params, f)

    def load_params(self, file_name='Rnnlm.pkl'):
        with open(file_name, 'rb') as f:
            self.params = pickle.load(f)
```

Rnnlm 类将到 Softmax 层为止的处理实现为 predict() 方法，这个方法在第 7 章进行文本生成时还会用到。此外，该类还添加了用于读写参数的 save_params() 和 load_params() 方法。剩下的实现与上一章的 SimpleRnnlm 类相同。



common/base_model.py 中有一个 BaseModel 类，该类实现了 save_params() 和 load_params() 方法。因此，通过继承 BaseModel 类，也能获得数据读写的功能。另外，BaseModel 类的实现还进行了优化，以支持 GPU 和进行缩位（使用 16 位浮点数存储）。

下面，我们在 PTB 数据集上学习这个网络。这次我们使用 PTB 数据集的所有训练数据进行学习（上一章中只使用了 PTB 数据集的一部分），代码如下所示（✏ ch06/train_rnnlm.py）。

```
import sys
sys.path.append('..')
from common.optimizer import SGD
from common.trainer import RnnlmTrainer
from common.util import eval_perplexity
```

```
from dataset import ptb
from rnnlm import Rnnlm

# 设定超参数
batch_size = 20
wordvec_size = 100
hidden_size = 100 # RNN的隐藏状态向量的元素个数
time_size = 35 # RNN的展开大小
lr = 20.0
max_epoch = 4
max_grad = 0.25

# 读入训练数据
corpus, word_to_id, id_to_word = ptb.load_data('train')
corpus_test, _, _ = ptb.load_data('test')
vocab_size = len(word_to_id)
xs = corpus[:-1]
ts = corpus[1:]

# 生成模型
model = Rnnlm(vocab_size, wordvec_size, hidden_size)
optimizer = SGD(lr)
trainer = RnnlmTrainer(model, optimizer)

# ❶应用梯度裁剪进行学习
trainer.fit(xs, ts, max_epoch, batch_size, time_size, max_grad,
            eval_interval=20)
trainer.plot(ylim=(0, 500))

# ❷基于测试数据进行评价
model.reset_state()
ppl_test = eval_perplexity(model, corpus_test)
print('test perplexity: ', ppl_test)

# ❸保存参数
model.save_params()
```

这里给出的代码和上一章的代码（✏ ch05/train.py）有很多相同的地方，因此这里重点介绍不同的地方。首先，代码❶处使用 RnnlmTrainer 类进行模

型的学习。RnnlmTrainer 类的 fit() 方法求模型的梯度，更新模型的参数。另外，在方法内部，通过指定 max_grad 参数，从而应用梯度裁剪。顺便说一下，fit() 方法内部进行的实现如下所示（这里给出的是伪代码）。

```
# 求梯度
model.forward(...)
model.backward(...)
params, grads = model.params, model.grads
# 梯度裁剪
if max_grad is not None:
    clip_grads(grads, max_grad)
# 更新参数
optimizer.update(params, grads)
```

我们在 6.1.4 节将梯度裁剪实现为了 clip_grads(grads, max_grad)，这里使用该方法进行梯度裁剪。

另外，通过❶处的 fit() 方法的参数 eval_interval=20，每 20 次迭代对困惑度进行 1 次评价。因为这次的数据量很大，所以没有对每个 epoch 进行评价，而是每 20 次迭代评价 1 次。后面我们会将评价结果用 plot() 方法绘制成图。

在学习结束后，在代码❷处使用测试数据对困惑度进行评价。这里需要注意的是，此时需要先重置模型的状态（LSTM 的隐藏状态和记忆单元）。此外，因为评价困惑度的函数 eval_perplexity() 在 common/util.py 中已经实现，所以直接使用即可。

最后，在代码❸处将学习好的参数保存到外部文件。在下一章生成句子时，将会使用这些学习好的权重参数。

以上就是 RNNLM 的学习代码。执行代码后，在终端上会输出图 6-27 的结果。

```
$python train_rnnlm.py
| epoch 1 |  iter 1 / 1327 | time 0[s] | perplexity 10000.84
| epoch 1 |  iter 21 / 1327 | time 4[s] | perplexity 3065.17
| epoch 1 |  iter 41 / 1327 | time 9[s] | perplexity 1255.96
| epoch 1 |  iter 61 / 1327 | time 14[s] | perplexity 956.13
| epoch 1 |  iter 81 / 1327 | time 18[s] | perplexity 806.56
| epoch 1 |  iter 101 / 1327 | time 23[s] | perplexity 658.86
| epoch 1 |  iter 121 / 1327 | time 27[s] | perplexity 636.88
| epoch 1 |  iter 141 / 1327 | time 31[s] | perplexity 601.75
| epoch 1 |  iter 161 / 1327 | time 35[s] | perplexity 575.78
| epoch 1 |  iter 181 / 1327 | time 40[s] | perplexity 590.01
| epoch 1 |  iter 201 / 1327 | time 44[s] | perplexity 479.95
| epoch 1 |  iter 221 / 1327 | time 48[s] | perplexity 488.23
| epoch 1 |  iter 241 / 1327 | time 53[s] | perplexity 443.62
| epoch 1 |  iter 261 / 1327 | time 57[s] | perplexity 468.75
| epoch 1 |  iter 281 / 1327 | time 61[s] | perplexity 447.81
| epoch 1 |  iter 301 / 1327 | time 66[s] | perplexity 398.51
| epoch 1 |  iter 321 / 1327 | time 70[s] | perplexity 350.89
| epoch 1 |  iter 341 / 1327 | time 74[s] | perplexity 406.82
| epoch 1 |  iter 361 / 1327 | time 79[s] | perplexity 409.33
| epoch 1 |  iter 381 / 1327 | time 83[s] | perplexity 337.04
```

图6-27 终端的输出结果

在图 6-27 中，每 20 次迭代输出 1 次困惑度的值。我们来看一下结果，刚开始的困惑度为 10 000.84，这意味着下一个单词的候选个数能减少到 10 000 个左右。因为这次数据集的词汇量是 10 000 个，所以这是什么也没学习的状态，相当于猜测。但是随着学习的进行，困惑度开始变好。实际上，当迭代超过 300 次时，困惑度已经降到了 400 以下。现在，我们看一下困惑度的演变图，如图 6-28 所示。

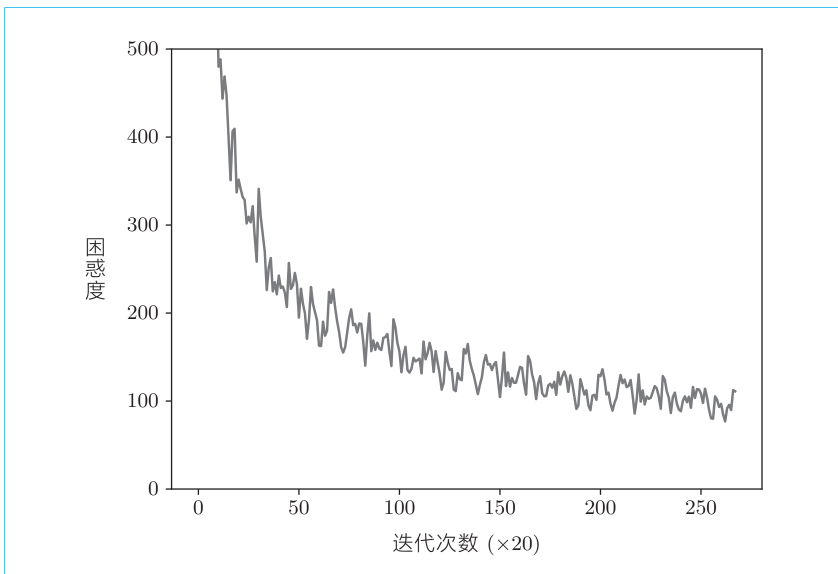


图 6-28　困惑度的演变（每 20 次迭代对训练数据进行 1 次评价）

在这次的实验中，一共进行了 4 个 epoch 的学习（按迭代来算，相当于 1327 * 4 次）。如图 6-28 所示，困惑度顺利下降，最终达到 100 左右。基于最终的测试数据的评价（源代码❷处）结果为 136.07...。该结果在每次执行时都不相同，但是都在 135 前后。换句话说，我们的模型成长到了能将下一个单词的候选个数（从 10 000 个）缩小到 136 个左右的水平。

那么，136 这样的困惑度在实践中是什么水平呢？说实话，这并不是一个很好的结果。在 2017 年的一个研究中，PTB 数据集上的困惑度已经降到了 60 以下 [34]。我们的模型还有很大的改进空间，下面我们就来进一步改进现有的 RNNLM。

6.5　进一步改进 RNNLM

本节我们先针对当前的 RNNLM 说明 3 点需要改进的地方，然后实施这些改进，并评价最后精度提高了多少。

6.5.1　LSTM 层的多层化

在使用 RNNLM 创建高精度模型时，加深 LSTM 层（叠加多个 LSTM 层）的方法往往很有效。之前我们只用了一个 LSTM 层，通过叠加多个层，可以提高语言模型的精度。例如，在图 6-29 中，RNNLM 使用了两个 LSTM 层。

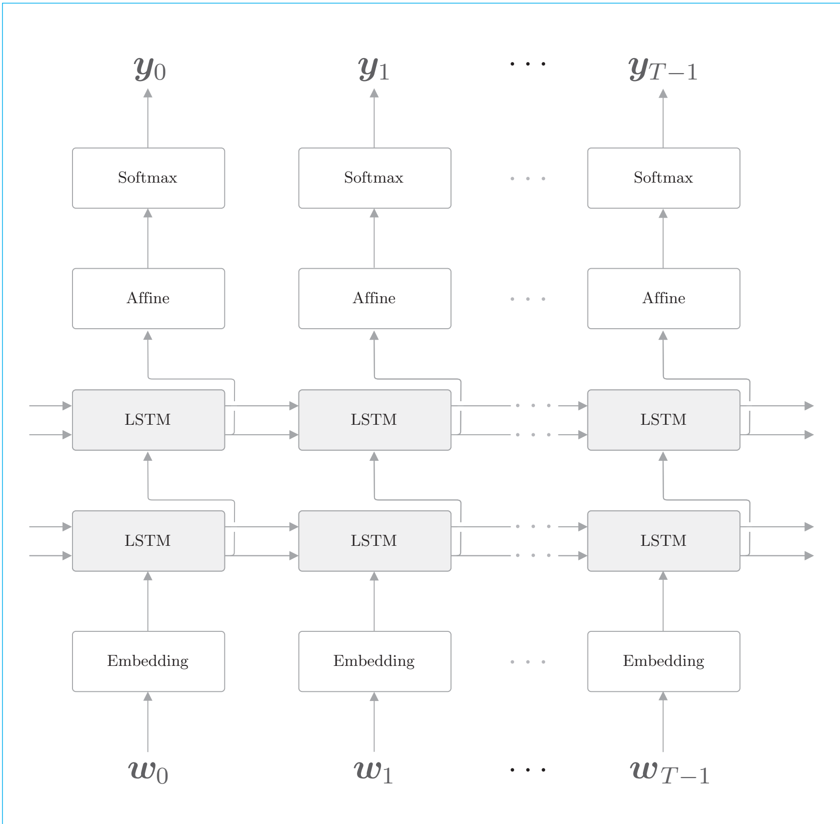


图 6-29　使用两个 LSTM 层的 RNNLM

图 6-29 显示了叠加两个 LSTM 层的例子。此时，第一个 LSTM 层的隐藏状态是第二个 LSTM 层的输入。按照同样的方式，我们可以叠加多个

LSTM 层，从而学习更加复杂的模式，这和前馈神经网络时的层加深是一样的。在前作《深度学习入门：基于 Python 的理论与实现》中，我们通过叠加多个 Affine 层和 Convolution 层，创建了表现力更好的模型。

那么，应该叠加几个层呢？这其实是一个关于超参数的问题。因为层数是超参数，所以需要根据要解决的问题的复杂程度、能给到的训练数据的规模来确定。顺便说一句，在 PTB 数据集上学习语言模型的情况下，当 LSTM 的层数为 2 ～ 4 时，可以获得比较好的结果。



据报道，谷歌翻译中使用的 GNMT 模型[50]是叠加了 8 层 LSTM 的网络。如该例所示，如果待解决的问题很难，又能准备大量的训练数据，就可以通过加深 LSTM 层来提高精度。

6.5.2　基于 Dropout 抑制过拟合

通过叠加 LSTM 层，可以期待能够学习到时序数据的复杂依赖关系。换句话说，通过加深层，可以创建表现力更强的模型，但是这样的模型往往会发生过拟合（overfitting）。更糟糕的是，RNN 比常规的前馈神经网络更容易发生过拟合，因此 RNN 的过拟合对策非常重要。



过拟合是指过度学习了训练数据的状态，也就是说，过拟合是一种缺乏泛化能力的状态。我们想要的是一个泛化能力强的模型，因此必须基于训练数据和验证数据的评价差异，判断是否发生了过拟合，并据此来进行模型的设计。

抑制过拟合已有既定的方法：一是增加训练数据；二是降低模型的复杂度。我们会优先考虑这两个方法。除此之外，对模型复杂度给予惩罚的正则化也很有效。比如，L2 正则化会对过大的权重进行惩罚。

此外，像 Dropout[9] 这样，在训练时随机忽略层的一部分（比如 50%）神经元，也可以被视为一种正则化（图 6-30）。本节我们将仔细研究 Dropout，并将其应用于 RNN。

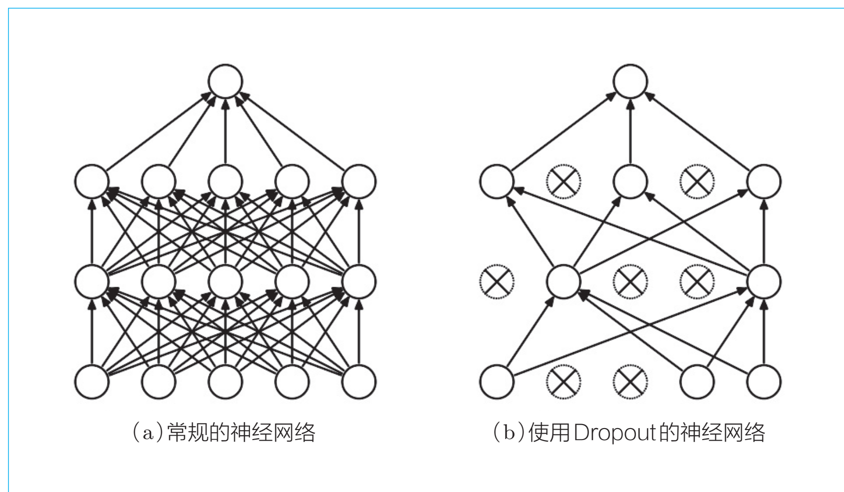


图6-30 Dropout的概念图(参考文献[9]):左边是常规的神经网络，右边是使用了Dropout的网络

如图 6-30 所示，Dropout 随机选择一部分神经元，然后忽略它们，停止向前传递信号。这种“随机忽视”是一种制约，可以提高神经网络的泛化能力。我们在前作《深度学习入门：基于 Python 的理论与实现》中已经实现了 Dropout。如图 6-31 所示，当时我们给出了在激活函数后插入 Dropout 层的示例，并展示了它有助于抑制过拟合。

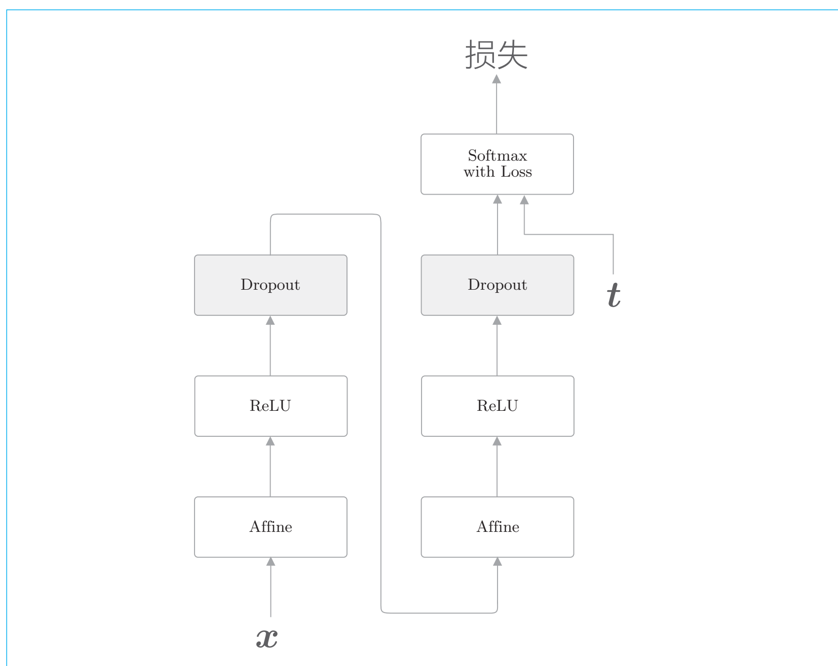


图 6-31 将 Dropout 层应用于前馈神经网络的例子

那么，在使用 RNN 的模型中，应该将 Dropout 层插入哪里呢？首先可以想到的是插入在 LSTM 层的时序方向上，如图 6-32 所示。不过答案是，这并不是一个好的插入方式。

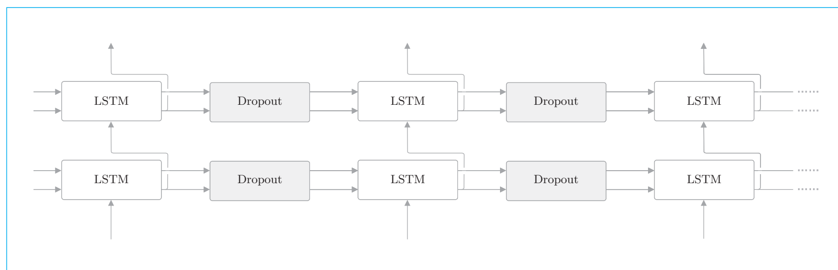


图 6-32 不好的例子：在时序方向上插入 Dropout 层

如果在时序方向上插入 Dropout，那么当模型学习时，随着时间的推

移，信息会渐渐丢失。也就是说，因 Dropout 产生的噪声会随时间成比例地积累。考虑到噪声的积累，最好不要在时间轴方向上插入 Dropout。因此，如图 6-33 所示，我们在深度方向（垂直方向）上插入 Dropout 层。

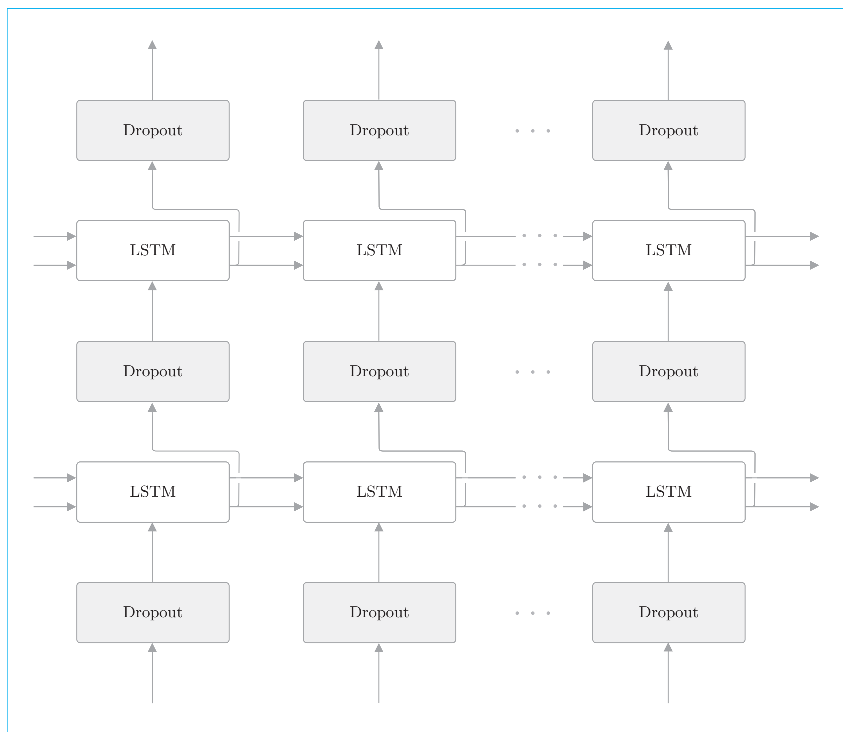


图6-33 好的例子：在深度方向（垂直方向）上插入Dropout层

这样一来，无论沿时间方向（水平方向）前进多少，信息都不会丢失。Dropout 与时间轴独立，仅在深度方向（垂直方向）上起作用。



现在比较一下图6-31和图6-33。图6-31的例子展示了在前馈神经网络中使用Dropout的情况，这个例子在深度方向上应用了Dropout。以相同的方式，在图6-33中，通过在深度方向上应用Dropout，有望和前馈神经网络时一样，能够抑制过拟合。

如前所述，“常规的 Dropout”不适合用在时间方向上。但是，最近的研究提出了多种方法来实现时间方向上的 RNN 正则化。比如，文献 [36] 中提出的“变分 Dropout”（variational dropout）就被成功地应用在了时间方向上。

除了深度方向，变分 Dropout 也能用在时间方向上，从而进一步提高语言模型的精度。如图 6-34 所示，它的机制是同一层的 Dropout 使用相同的 mask。这里所说的 mask 是指决定是否传递数据的随机布尔值。

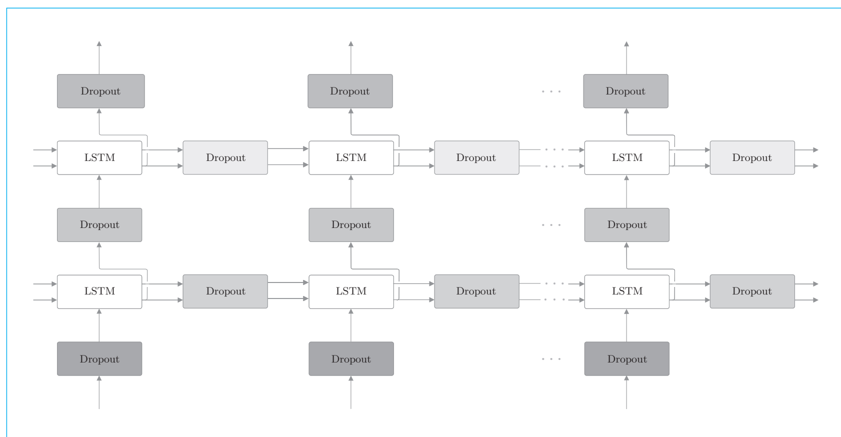


图6-34 变分 Dropout 的例子：具有相同图示的 Dropout 使用相同的 mask。像这样，位于同一层的 Dropout 使用相同的 mask，对时间方向上的 Dropout 也有效果（参见彩图）

如图 6-34 所示，通过同一层的 Dropout 共用 mask，mask 被“固定”。如此一来，信息的损失方式也被“固定”，所以可以避免常规 Dropout 发生的指数级信息损失。



据说变分 Dropout 比常规 Dropout 的效果更好。不过，本章并不打算使用变分 Dropout，而是仍使用常规 Dropout。变分 Dropout 的想法很简单，感兴趣的读者可以自己尝试实现一下。

6.5.3 权重共享

改进语言模型有一个非常简单的技巧，那就是权重共享（weight tying）[37][38]。weight tying 可以直译为“权重绑定”。如图 6-35 所示，其含义就是共享权重。

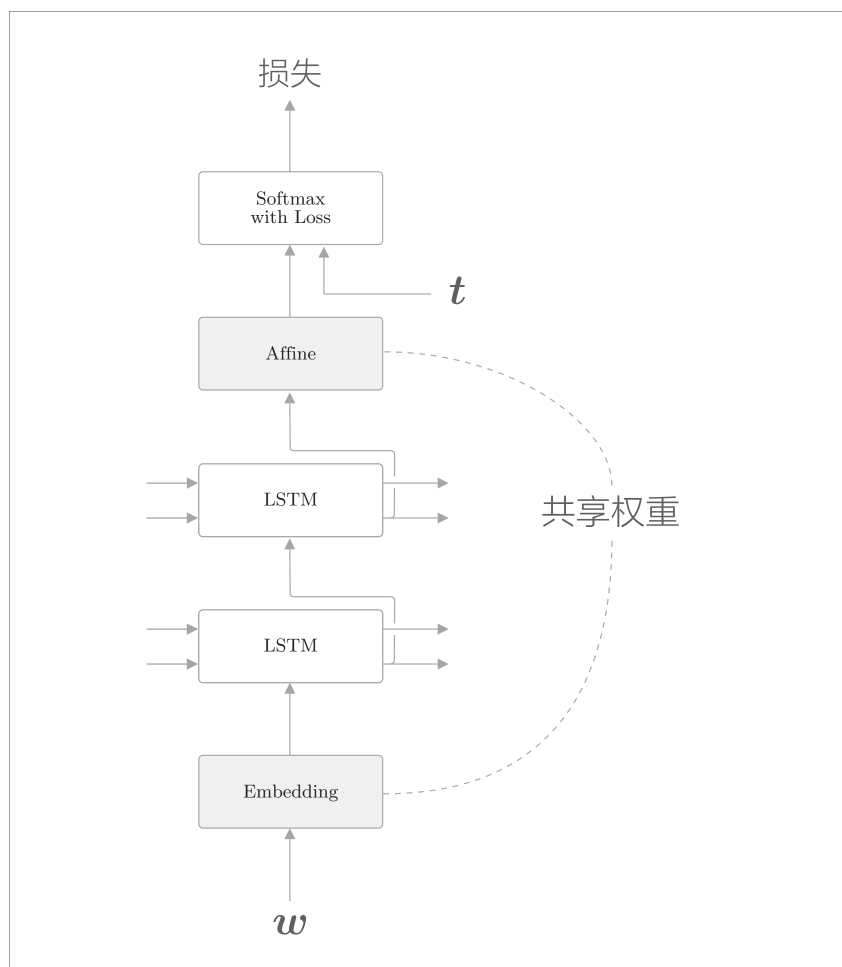


图6-35 语言模型中共享权重的例子：Embedding层和Softmax前的Affine层共享权重

如图 6-35 所示，绑定（共享）Embedding 层和 Affine 层的权重的技巧在于权重共享。通过在这两个层之间共享权重，可以大大减少学习的参数数量。尽管如此，它仍能提高精度。真可谓一石二鸟！

现在，我们来考虑一下权重共享的实现。这里，假设词汇量为 V，LSTM 的隐藏状态的维数为 H，则 Embedding 层的权重形状为 $V \times H$，Affine 层的权重形状为 $H \times V$。此时，如果要使用权重共享，只需将 Embedding 层权重的转置设置为 Affine 层的权重。这个非常简单的技巧可以带来出色的结果。



为什么说权重共享是有效的呢？直观上，共享权重可以减少需要学习的参数数量，从而促进学习。另外，参数数量减少，还能收获抑制过拟合的好处。论文 [38] 从理论上描述了权重共享为什么有用，感兴趣的读者可以参考一下。

6.5.4 更好的 RNNLM 的实现

至此，我们介绍了 RNNLM 的 3 点有待改进的地方。接下来，我们来看一下这些技巧会在多大程度上有效。这里，将图 6-36 的层结构实现为 `BetterRnnlm` 类。

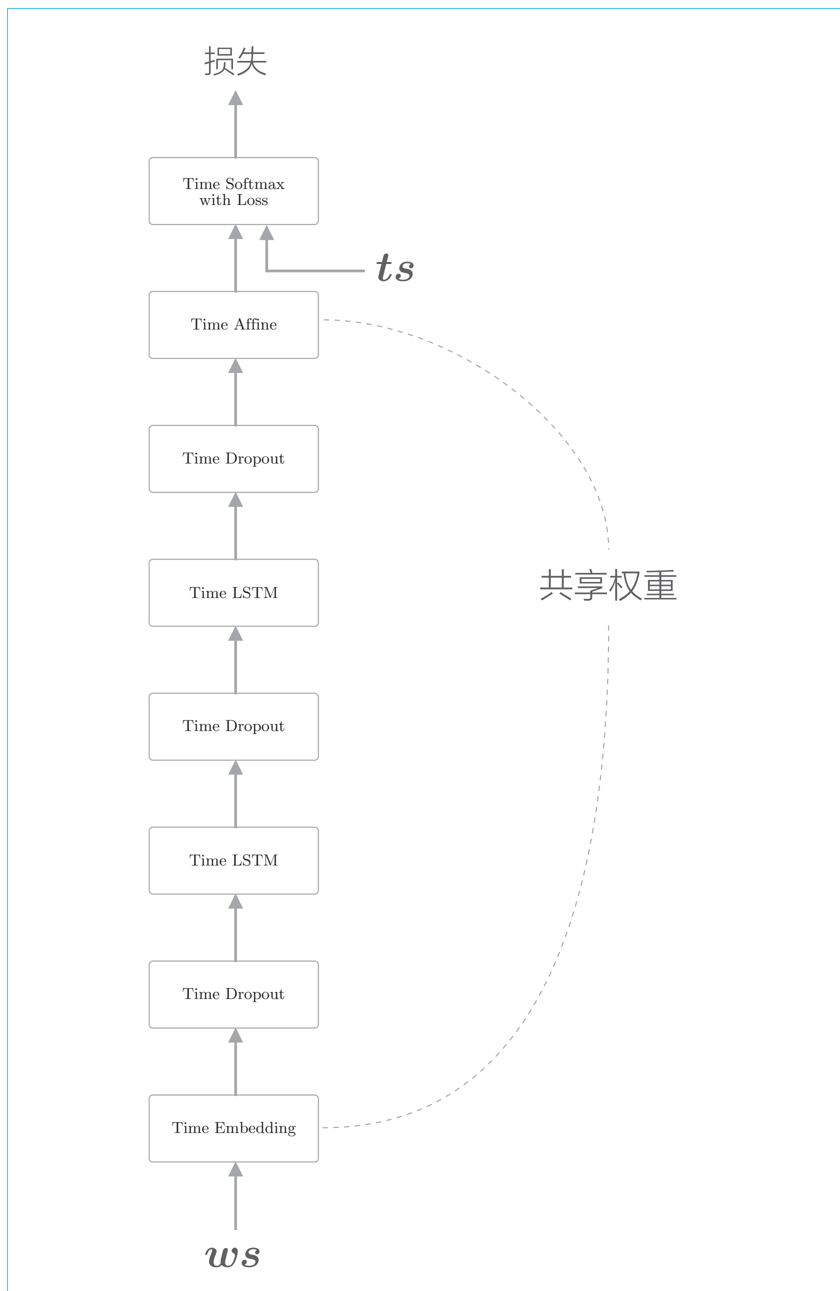


图6-36 BetterRnnlm类的网络结构

如图 6-36 所示，改进的 3 点如下：

- LSTM 层的多层化（此处为 2 层）
- 使用 Dropout（仅应用在深度方向上）
- 权重共享（Embedding 层和 Affine 层的权重共享）

现在，我们来实现进行了这 3 点改进的 BetterRnnlm 类，如下所示（✏ ch06/better_rnnlm.py）。

```
import sys
sys.path.append('..')
from common.time_layers import *
from common.np import *
from common.base_model import BaseModel

class BetterRnnlm(BaseModel):
    def __init__(self, vocab_size=10000, wordvec_size=650,
                 hidden_size=650, dropout_ratio=0.5):
        V, D, H = vocab_size, wordvec_size, hidden_size
        rn = np.random.randn

        embed_W = (rn(V, D) / 100).astype('f')
        lstm_Wx1 = (rn(D, 4 * H) / np.sqrt(D)).astype('f')
        lstm_Wh1 = (rn(H, 4 * H) / np.sqrt(H)).astype('f')
        lstm_b1 = np.zeros(4 * H).astype('f')
        lstm_Wx2 = (rn(H, 4 * H) / np.sqrt(H)).astype('f')
        lstm_Wh2 = (rn(H, 4 * H) / np.sqrt(H)).astype('f')
        lstm_b2 = np.zeros(4 * H).astype('f')
        affine_b = np.zeros(V).astype('f')

        # 3点改进!
        self.layers = [
            TimeEmbedding(embed_W),
            TimeDropout(dropout_ratio),
            TimeLSTM(lstm_Wx1, lstm_Wh1, lstm_b1, stateful=True),
            TimeDropout(dropout_ratio),
            TimeLSTM(lstm_Wx2, lstm_Wh2, lstm_b2, stateful=True),
            TimeDropout(dropout_ratio),
            TimeAffine(embed_W.T, affine_b) # 权重共享!!
```

```
        ]
        self.loss_layer = TimeSoftmaxWithLoss()
        self.lstm_layers = [self.layers[2], self.layers[4]]
        self.drop_layers = [self.layers[1], self.layers[3],
↪   self.layers[5]]
        self.params, self.grads = [], []
        for layer in self.layers:
            self.params += layer.params
            self.grads += layer.grads

    def predict(self, xs, train_flg=False):
        for layer in self.drop_layers:
            layer.train_flg = train_flg
        for layer in self.layers:
            xs = layer.forward(xs)
        return xs

    def forward(self, xs, ts, train_flg=True):
        score = self.predict(xs, train_flg)
        loss = self.loss_layer.forward(score, ts)
        return loss

    def backward(self, dout=1):
        dout = self.loss_layer.backward(dout)
        for layer in reversed(self.layers):
            dout = layer.backward(dout)
        return dout

    def reset_state(self):
        for layer in self.lstm_layers:
            layer.reset_state()
```

灰色背景的代码就是刚才所说的改进的地方，具体而言，叠加两个 Time LSTM 层，使用 Time Dropout 层，并在 Time Embedidng 层和 Time Affine 层之间共享权重。

下面进行改进过的 BetterRnnlm 类的学习。在这之前，我们稍微改动一下将要执行的学习代码。这个改动是，针对每个 epoch 使用验证数据评价困惑度，在值变差时，降低学习率。这是一种在实践中经常用到的技巧，并且

往往能有好的结果。这里的实现参考了 PyTorch 的语言模型的实现示例[39]，学习代码如下所示（✏ ch06/train_better_rnnlm.py）。

```
import sys
sys.path.append('..')
from common import config
# 在用GPU运行时，请打开下面的注释（需要cupy）
# ==============================================
# config.GPU = True
# ==============================================
from common.optimizer import SGD
from common.trainer import RnnlmTrainer
from common.util import eval_perplexity
from dataset import ptb
from better_rnnlm import BetterRnnlm


# 设定超参数
batch_size = 20
wordvec_size = 650
hidden_size = 650
time_size = 35
lr = 20.0
max_epoch = 40
max_grad = 0.25
dropout = 0.5

# 读入训练数据
corpus, word_to_id, id_to_word = ptb.load_data('train')
corpus_val, _, _ = ptb.load_data('val')
corpus_test, _, _ = ptb.load_data('test')

vocab_size = len(word_to_id)
xs = corpus[:-1]
ts = corpus[1:]

model = BetterRnnlm(vocab_size, wordvec_size, hidden_size, dropout)
optimizer = SGD(lr)
trainer = RnnlmTrainer(model, optimizer)
```

```
best_ppl = float('inf')
for epoch in range(max_epoch):
    trainer.fit(xs, ts, max_epoch=1, batch_size=batch_size,
                time_size=time_size, max_grad=max_grad)

    model.reset_state()
    ppl = eval_perplexity(model, corpus_val)
    print('valid perplexity: ', ppl)

    if best_ppl > ppl:
        best_ppl = ppl
        model.save_params()
    else:
        lr /= 4.0
        optimizer.lr = lr
    model.reset_state()
    print('-' * 50)
```

这里针对每个 epoch 使用验证数据评价困惑度，当它比之前的困惑度（`best_ppl`）低时，将学习率乘以 1/4。为此，我们用 `for` 循环反复执行以下处理：通过 `RnnlmTrainer` 类的 `fit()` 方法进行一个 epoch 的学习，然后使用验证数据评价困惑度。现在让我们运行一下学习代码。



这个学习需要相当长的时间。在用CPU运行的情况下，需要2天左右；而如果用GPU运行，则能在5小时左右完成（在用GPU运行时，需要去掉文件顶部`import`语句中的`# config.GPU = True`这行注释）。此外，从出版社网站的本书主页可以获得学习好的权重。

执行上面的代码，困惑度平稳下降，最终在测试数据上获得了困惑度为 75.76 的结果（每次运行结果不同）。考虑到改进前的 RNNLM 的困惑度约为 136，这个结果可以说提升很大。通过 LSTM 的多层化提高表现力，通过 Dropout 提高泛化能力，通过权重共享有效利用权重，从而实现了精度的大幅提高。

6.5.5 前沿研究

至此，我们对 RNNLM 的改进就结束了。通过对 RNNLM 进行若干改造，精度显著提升，在 PTB 数据集的测试数据上达到了 75 左右的困惑度，可以说是一个还算不错的结果。不过，前沿研究走得更远。这里我想简单地介绍一下最新的研究结果，让我们来看一下图 6-37。

Model	Parameters	Validation	Test
Mikolov & Zweig (2012) - KN-5	2M†	–	141.2
Mikolov & Zweig (2012) - KN5 + cache	2M‡	–	125.7
Mikolov & Zweig (2012) - RNN	6M‡	–	124.7
Mikolov & Zweig (2012) - RNN-LDA	7M‡	–	113.7
Mikolov & Zweig (2012) - RNN-LDA + KN-5 + cache	9M‡	–	92.0
Zaremba et al. (2014) - LSTM (medium)	20M	86.2	82.7
Zaremba et al. (2014) - LSTM (large)	66M	82.2	78.4
Gal & Ghahramani (2016) - Variational LSTM (medium)	20M	81.9 ± 0.2	79.7 ± 0.1
Gal & Ghahramani (2016) - Variational LSTM (medium, MC)	20M	–	78.6 ± 0.1
Gal & Ghahramani (2016) - Variational LSTM (large)	66M	77.9 ± 0.3	75.2 ± 0.2
Gal & Ghahramani (2016) - Variational LSTM (large, MC)	66M	–	73.4 ± 0.0
Kim et al. (2016) - CharCNN	19M	–	78.9
Merity et al. (2016) - Pointer Sentinel-LSTM	21M	72.4	70.9
Grave et al. (2016) - LSTM	–	–	82.3
Grave et al. (2016) - LSTM + continuous cache pointer	–	–	72.1
Inan et al. (2016) - Variational LSTM (tied) + augmented loss	24M	75.7	73.2
Inan et al. (2016) - Variational LSTM (tied) + augmented loss	51M	71.1	68.5
Zilly et al. (2016) - Variational RHN (tied)	23M	67.9	65.4
Zoph & Le (2016) - NAS Cell (tied)	25M	–	64.0
Zoph & Le (2016) - NAS Cell (tied)	54M	–	62.4
Melis et al. (2017) - 4-layer skip connection LSTM (tied)	24M	60.9	58.3
AWD-LSTM - 3-layer LSTM (tied)	24M	60.0	57.3
AWD-LSTM - 3-layer LSTM (tied) + continuous cache pointer	24M	53.9	52.8

图 6-37 各模型在 PTB 数据集上的结果（摘自文献 [34]）。表中的 Parameters 是参数总数，Validation 是验证数据的困惑度，Test 是测试数据的困惑度

图 6-37 摘自文献 [34]，该表总结了过去各个阶段最优语言模型在 PTB 数据集上的困惑度结果。由 Test 列可知，随着新方法被提出，困惑度在下降，最后一行的结果是 52.8。实际上，这个 52.8 是一个非常好的结果。在 PTB 数据集上的困惑度接近 50，这在几年前还是无法想象的。

这里只展示了最先进的研究结果。当然，我们的模型和它还有相当的距离，但是图 6-37 中的最先进的模型和我们的模型有很多共同点。比如，最

先进的模型使用了多层 LSTM 模型，并进行了基于 Dropout 的正则化（变分 Dropout 和 DropConnect[①]）和权重共享。在此基础上，它进一步使用了最优化和正则化的几个技巧，并严格进行了超参数的调整，最终达成了 52.8 这样惊人的值。



图6-37中有一个名为AWD-LSTM 3-layer LSTM (tied) + continuous cache pointer的模型。这个continuous cache pointer技术基于第8章会详细介绍的Attention。Attention是一项非常重要的技术，被应用在许多地方。在语言模型这个任务中，它也为精度提高做出了重大贡献。让我们期待第8章的Attention。

6.6 小结

本章的主题是 Gated RNN，我们指出了上一章的简单 RNN 中存在的梯度消失（或梯度爆炸）问题，说明了作为替代层的 Gated RNN（具体指 LSTM 和 GRU 等）的有效性。这些层使用门这一机制，能够更好地控制数据和梯度的流动。

另外，本章使用 LSTM 层创建了语言模型，并在 PTB 数据集上进行了学习，评价了困惑度。另外，通过 LSTM 的多层化、Dropout 和权重共享等技巧，成功地大幅提高了精度。这些技巧也被实际用在了 2017 年的最前沿研究中。

下一章我们将使用语言模型生成文本。之后，像机器翻译一样，我们将仔细考察一个将某种语言转换为另一种语言的模型。

① DropConnect是指随机无视权重自身的方法。

本章所学的内容

- 在简单 RNN 的学习中，存在梯度消失和梯度爆炸问题
- 梯度裁剪对解决梯度爆炸有效，LSTM、GRU 等 Gated RNN 对解决梯度消失有效
- LSTM 中有 3 个门：输入门、遗忘门和输出门
- 门有专门的权重，并使用 sigmoid 函数输出 0.0 ~ 1.0 的实数
- LSTM 的多层化、Dropout 和权重共享等技巧可以有效改进语言模型
- RNN 的正则化很重要，人们提出了各种基于 Dropout 的方法

第 7 章
基于 RNN 生成文本

不存在什么完美的文章，就好像没有完美的绝望。
——村上春树《且听风吟》

在第 5 章和第 6 章中，我们仔细研究了 RNN 和 LSTM 的结构及其实现。现在我们已经在代码层面理解了它们。在本章，RNN 和 LSTM 将大显身手，我们将利用 LSTM 实现几个有趣的应用。

首先，本章将使用语言模型进行文本生成。具体来说，就是使用在语料库上训练好的语言模型生成新的文本。然后，我们将了解如何使用改进过的语言模型生成更加自然的文本。通过这项工作，我们可以（简单地）体验基于 AI 的文本创作。

另外，本章还会介绍一种结构名为 seq2seq 的新神经网络。seq2seq 是“(from) sequence to sequence”（从时序到时序）的意思，即将一个时序数据转换为另一个时序数据。本章我们将看到，通过组合两个 RNN，可以轻松实现 seq2seq。seq2seq 可以应用于多个应用，比如机器翻译、聊天机器人和邮件自动回复等。通过理解这个简单但聪明强大的 seq2seq，应用深度学习的可能性将进一步扩大。

7.1 使用语言模型生成文本

我们已经用几章的篇幅讨论了语言模型。如前所述，语言模型可用于各种各样的应用，其中具有代表性的例子有机器翻译、语音识别和文本生成。这里，我们将使用语言模型来生成文本。

7.1.1 使用RNN生成文本的步骤

在上一章中，我们使用 LSTM 层实现了语言模型，这个语言模型的网络结构如图 7-1 所示。顺便说一下，我们还实现了整体处理（T个）时序数据的 Time LSTM 层和 Time Affine 层。

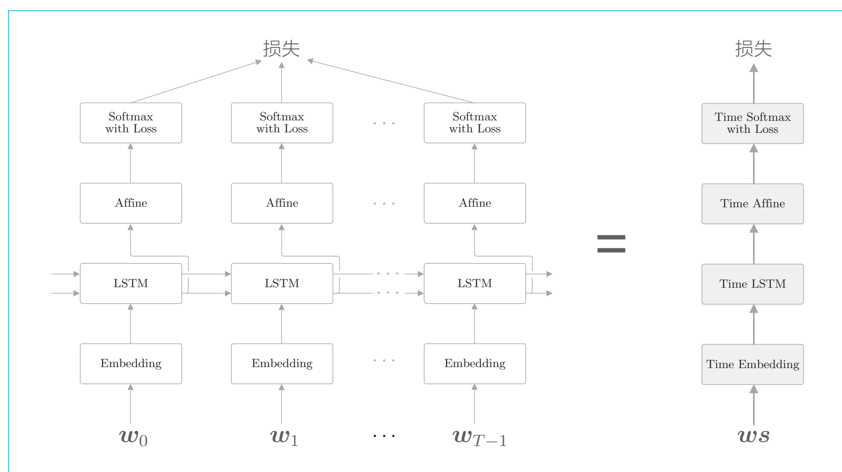


图 7-1 上一章实现的语言模型：右图使用整体处理时序数据的 Time 层；左图是将其展开后的层结构

现在我们来说明一下语言模型生成文本的顺序。这里仍以“you say goobye and i say hello.”这一在语料库上学习好的语言模型为例，考虑将单词 i 赋给这个语言模型的情况。此时，这个语言模型输出图 7-2 中的概率分布。

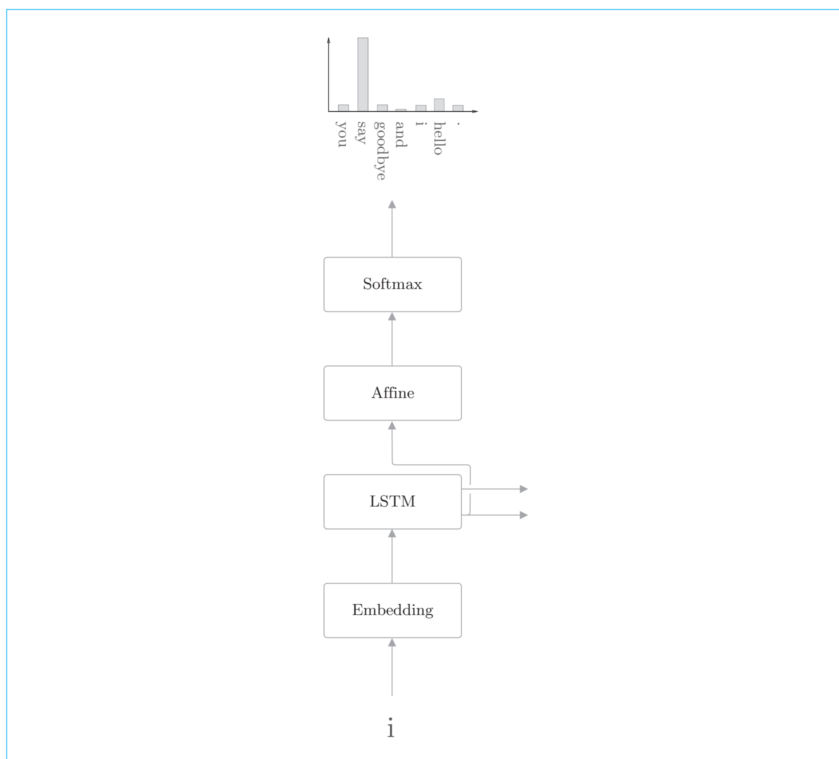


图 7-2 语言模型输出下一个出现的单词的概率分布

语言模型根据已经出现的单词输出下一个出现的单词的概率分布。在图 7-2 的例子中，语言模型输出了当给定单词 i 时下一个出现的单词的概率分布。那么，它如何生成下一个新单词呢？

一种可能的方法是选择概率最高的单词。在这种情况下，因为选择的是概率最高的单词，所以结果能唯一确定。也就是说，这是一种“确定性的”方法。另一种方法是“概率性地”进行选择。根据概率分布进行选择，这样概率高的单词容易被选到，概率低的单词难以被选到。在这种情况下，被选到的单词（被采样到的单词）每次都不一样。

这里我们想让每次生成的文本有所不同，这样一来，生成的文本富有变化，会更有趣。因此，我们通过后一种方法（概率性地选择的方法）来

选择单词。回到我们的例子中，如图 7-3 所示，假设（概率性地）选择了单词 say。

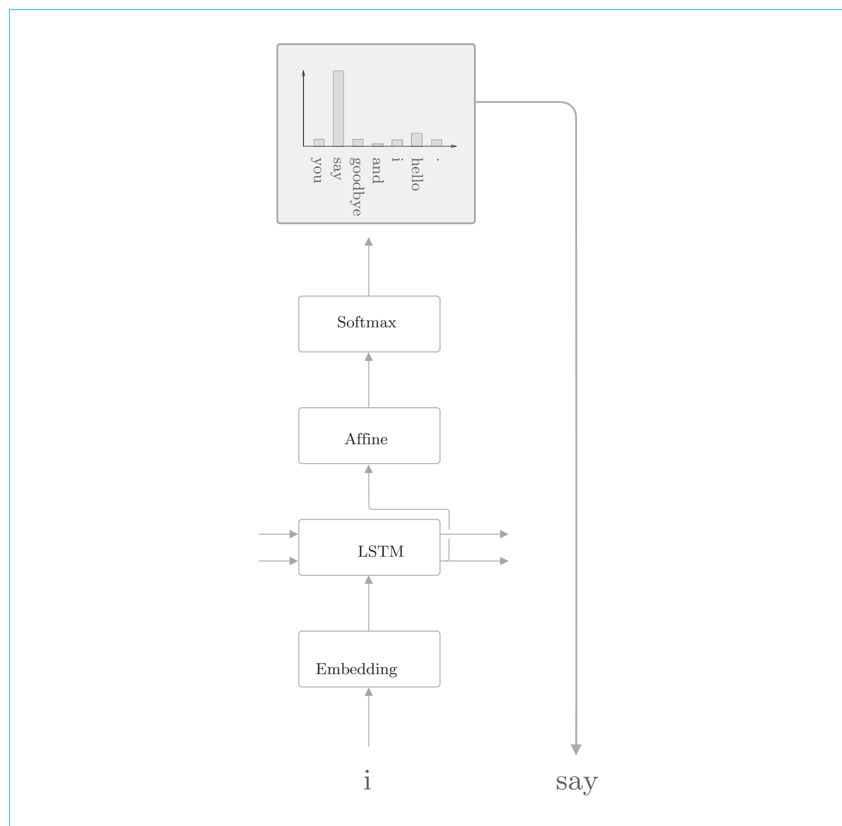


图 7-3　根据概率分布采样一个单词

图 7-3 中显示了根据概率分布进行采样后结果为 say 的例子。在图 7-3 的概率分布中，say 的概率最高，所以它被采样到的概率也最高。不过请注意，这里选到 say 并不是必然的（不是确定性的），而是概率性的。因此，say 以外的其他单词根据出现的概率也可能被采样到。



“确定性的”是指（算法的）结果是唯一确定的，是可预测的。在上例中，假设选择概率最高的单词，那么这就是一种确定性的算法。而“概率性的”算法则概率性地确定结果，因此每次实验时选到的单词都会有所变化（或者说，存在变化的可能性）。

接下来，采样第 2 个单词。这只需要重复一下刚才的操作。也就是说，将生成的单词 say 输入语言模型，获得单词的概率分布，然后再根据这个概率分布采样下一个出现的单词，如图 7-4 所示。

图 7-4　重复概率分布的输出和采样

之后根据需要重复此过程即可（或者直到出现 <eos> 这一结尾记号）。

这样一来，我们就可以生成新的文本。

这里需要注意的是，像上面这样生成的新文本是训练数据中没有的新生成的文本。因为语言模型并不是背诵了训练数据，而是学习了训练数据中单词的排列模式。如果语言模型通过语料库正确学习了单词的出现模式，我们就可以期待该语言模型生成的文本对人类而言是自然的、有意义的。

7.1.2　文本生成的实现

下面我们进行文本生成的实现。这里基于上一章实现的 Rnnlm 类（✏ ch06/rnnlm.py），来创建继承自它的 RnnlmGen 类，然后向这个类添加生成文本的方法。



类的继承是指继承已有类，创建新的类。在Python中，可以通过class New(Base):继承Base类，创建New类。

RnnlmGen 类的实现如下所示（✏ ch07/rnnlm_gen.py）。

```
import sys
sys.path.append('..')
import numpy as np
from common.functions import softmax
from ch06.rnnlm import Rnnlm
from ch06.better_rnnlm import BetterRnnlm


class RnnlmGen(Rnnlm):
    def generate(self, start_id, skip_ids=None, sample_size=100):
        word_ids = [start_id]

        x = start_id
        while len(word_ids) < sample_size:
            x = np.array(x).reshape(1, 1)
            score = self.predict(x)
            p = softmax(score.flatten())

            sampled = np.random.choice(len(p), size=1, p=p)
```

```
            if (skip_ids is None) or (sampled not in skip_ids):
                x = sampled
                word_ids.append(int(x))

        return word_ids
```

这个类用 generate(start_id, skip_ids, sample_size) 生成本文。此处，参数 start_id 是第 1 个单词 ID，sample_size 表示要采样的单词数量。另外，参数 skip_ids 是单词 ID 列表（比如，[12, 20]），它指定的单词将不被采样。这个参数用于排除 PTB 数据集中的 <unk>、N 等被预处理过的单词。



PTB 数据集对原始文本进行了预处理，稀有单词被 <unk> 替换，数字被 N 替换。另外，我们用 <eos> 作为文本的分隔符。

generate() 方法首先通过 model.predict(x) 输出各个单词的得分（得分是正规化之前的值），然后基于 p = softmax(score)，使用 Softmax 函数对得分进行正规化，这样就获得了我们想要的概率分布。接下来，使用 np.random.choice()，根据这个概率分布 p 采样下一个单词。关于 np.random.choice()，我们已经在 4.2.6 节说明过了。



model 的 predict() 方法进行的是 mini-batch 处理，所以输入 x 必须是二维数组。因此，即使在只输入 1 个单词 ID 的情况下，也要将它的批大小视为 1，并将其整理成形状为 1 × 1 的 NumPy 数组。

现在，使用这个 RnnlmGen 类进行文本生成。这里先在完全没有学习的状态（即权重参数是随机初始值的状态）下生成文本，代码如下所示（✏ ch07/generate_text.py）。

```
import sys
sys.path.append('..')
from rnnlm_gen import RnnlmGen
```

```
from dataset import ptb


corpus, word_to_id, id_to_word = ptb.load_data('train')
vocab_size = len(word_to_id)
corpus_size = len(corpus)

model = RnnlmGen()
# model.load_params('../ch06/Rnnlm.pkl')

# 设定start单词和skip单词
start_word = 'you'
start_id = word_to_id[start_word]
skip_words = ['N', '<unk>', '$']
skip_ids = [word_to_id[w] for w in skip_words]

# 生成文本
word_ids = model.generate(start_id, skip_ids)
txt = ' '.join([id_to_word[i] for i in word_ids])
txt = txt.replace(' <eos>', '.\n')
print(txt)
```

这里，第 1 个单词是 you，我们将它的单词 ID 设为 start_id，来进行文本生成。另外，指定不参与采样的单词为 ['N', '<unk>', '$']。生成文本的 generate() 方法返回单词 ID 列表，因此需要将单词 ID 列表转化为句子。这可以通过 txt = ' '.join([id_to_word[i] for i in word_ids]) 这行代码进行。join() 方法通过“' 分隔符 '.join(列表)”这种形式连接单词。下面我们来看一个具体的例子。

```
>>> ' '.join(['you', 'say', 'goodbye'])
'you say goodbye'
```

运行一下上面的代码，结果如下。

```
you setback best raised fill steelworkers montgomery kohlberg told beam
worthy allied ban swedish aichi mather promptly ramada explicit leslie
bets discovery considering campaigns bottom petrie warm large-scale
frequent temple grumman bennett ...
```

如你所见，输出的文本是一堆乱七八糟的单词。不过这可以理解，因为这里的模型权重使用的是随机初始值，所以输出了没有意义的文本。那么，如果换成学习好的语言模型，结果会怎样呢？我们利用上一章学习好的权重来进行文本生成。为此，使用 `model.load_params('../ch06/Rnnlm.pkl')` 读入上一章学习好的权重参数，并生成文本。我们来看一下生成的文本（每次的结果都不一样）。

```
you 'll include one of them a good problems.
 moreover so if not gene 's corr experience with the heat of bridges a
new deficits model is non-violent what it 's a rule must exploit it.
 there 's no tires industry could occur.
 beyond my hours where he is n't going home says and japanese letter.
 knight transplants d.c. turmoil with one-third of voters.
 the justice department is ...
```

虽然上面的结果中可以看到多处语法错误和意思不通的地方，不过也有几处读起来已经比较像句子了。仔细看的话，这个模型正确生成了主语和动词的组合，比如“you'll include...”“there's no tires...”“knight transplants...”等。再者，它在一定程度上理解了形容词和名词的使用方法，比如“good problems”“japanese letter”等。另外，开头的“you'll include one of them a good problems.”也是一个含义通顺的句子。

如上所述，上面的实验生成的文本在某种程度上可以说是正确的，不过结果中仍有许多不自然的地方，改进空间很大。虽然不存在“完美的文章”，但是至少我们可以追求更自然的文章。为此，我们应该怎么做呢？当然是使用更好的语言模型！

7.1.3 更好的文本生成

如果有更好的语言模型，就可能有更好的文本。在上一章中，我们改进了简单的 RNNLM，实现了“更好的 RNNLM”，将模型的困惑度从 136 降至 75。现在，我们看一下这个“更好的 RNNLM”生成文本的能力。



在上一章中，我们进行了BetterRnnlm类的学习，并将学习好的权重保存为了文件。这里的实验需要用到这个学习好的权重文件，我们可以从出版社网站的本书主页获取。将这个权重文件放到本书源代码的ch06目录下，即可运行本实验的代码ch07/generate_better_text.py。

在上一章中，我们将更好的语言模型实现为了 BetterRnnlm 类。这里，像刚才一样，继承这个类，并使之有生成文本的能力。BetterRnnlmGen 类的实现和刚刚的 RnnlmGen 类完全一样，此处省略具体说明。

现在，我们让这个更好的语言模型生成文本。和之前一样，第 1 个单词是 you。这样一来，下述文本会被生成（ ✏ ch07/generate_better_text.py）。

```
you 've seen two families and the women and two other women of students.
 the principles of investors that prompted a bipartisan rule of which
had a withdrawn target of black men or legislators interfere with the
number of plants can do to carry it together.
 the appeal was to deny steady increases in the operation of dna and
educational damage in the 1950s.
...
```

可以看出，这个模型生成了比之前更自然的文本（可能有些主观）。最开始的句子“you' ve seen two families and the women...”正确使用了主语、动词和宾语，并且正确学习了 and 的使用方法（two families and the women）。其他部分读起来总体上也算说得过去。

虽然这里生成的文本仍然存在若干问题（特别是语义方面），但是从某种程度上来说，这个更好的语言模型生成了更加自然的文本。通过进一步改进这个模型，使用更大规模的语料库，应该能创造出更加自然的文本。

最后，我们尝试给这个更好的语言模型输入“the meaning of life is”，让它生成后续的单词（这是论文 [35] 中所做的实验）。为了做这个实验，我们按顺序向模型输入 ['the', 'meaning', 'of', 'life']，进行正向传播。此时不使用任何输出的结果，只是让 LSTM 层记住这些单词的信息。然后，以单词 is 作为开始位置，生成“the meaning of life is”的后续内容。

这个实验可以用 ch07/generate_better_text.py 实现。每次实验生成的文本都不太一样，这里介绍一个有意思的结果。

```
the meaning of life is not a good version of paintings.
```

如上所述，语言模型给出的回答是“人生的意义并不是一种状态良好的绘画”。虽然搞不懂什么才是“状态良好的绘画”，不过说不定这其中有什么深刻的意义。

7.2 seq2seq 模型

这个世界充满了时序数据。文本数据、音频数据和视频数据都是时序数据。另外，还存在许多需要将一种时序数据转换为另一种时序数据的任务，比如机器翻译、语音识别等。其他的还有进行对话的聊天机器人应用、将源代码转为机器语言的编译器等。

像这样，世界上存在许多输入输出均为时序数据的任务。从现在开始，我们会考察将时序数据转换为其他时序数据的模型。作为它的实现方法，我们将介绍使用两个 RNN 的 seq2seq 模型。

7.2.1 seq2seq 的原理

seq2seq 模型也称为 Encoder-Decoder 模型。顾名思义，这个模型有两个模块——Encoder（编码器）和 Decoder（解码器）。编码器对输入数据进行编码，解码器对被编码的数据进行解码。



编码是基于某些既定规则的信息转换过程。以字符码为例，将字符“A”转换为“1000001”（二进制）就是一个编码的例子。而解码则将被编码的信息还原到它的原始形态。仍以字符码为例，这相当于将位模式的“1000001”转换为字符“A”。

现在，我们举一个具体的例子来说明 seq2seq 的机制。这里考虑将日语翻译为英语，比如将“吾輩は猫である”[①] 翻译为“I am a cat”。此时，如图 7-5 所示，seq2seq 基于编码器和解码器进行时序数据的转换。

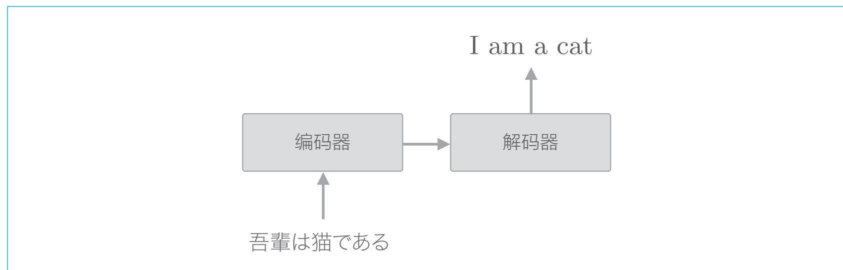


图 7-5　基于编码器和解码器进行翻译的例子

如图 7-5 所示，编码器首先对“吾輩は猫である”这句话进行编码，然后将编码好的信息传递给解码器，由解码器生成目标文本。此时，编码器编码的信息浓缩了翻译所必需的信息，解码器基于这个浓缩的信息生成目标文本。

以上就是 seq2seq 的全貌图。编码器和解码器协作，将一个时序数据转换为另一个时序数据。另外，在这些编码器和解码器内部可以使用 RNN。下面我们来看一下细节。首先来看编码器，它的层结构如图 7-6 所示。

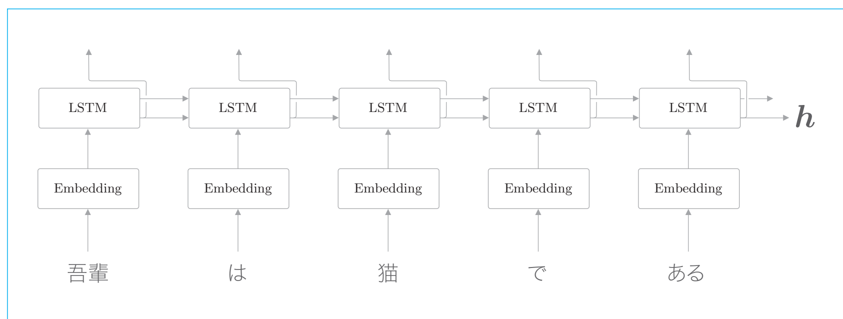


图 7-6　编码器的层结构

① “吾輩は猫である”是日本著名作家夏目漱石的代表作《我是猫》的书名，译为“我是猫”。——编者注

由图 7-6 可以看出，编码器利用 RNN 将时序数据转换为隐藏状态 $\boldsymbol{h}$。这里的RNN使用的是LSTM，不过也可以使用“简单RNN”或者GRU等。另外，这里考虑的是将日语句子分割为单词进行输入的情况。

图 7-6 的编码器输出的向量 $\boldsymbol{h}$ 是 LSTM 层的最后一个隐藏状态，其中编码了翻译输入文本所需的信息。这里的重点是，LSTM 的隐藏状态 $\boldsymbol{h}$ 是一个固定长度的向量。说到底，编码就是将任意长度的文本转换为一个固定长度的向量（图 7-7）。

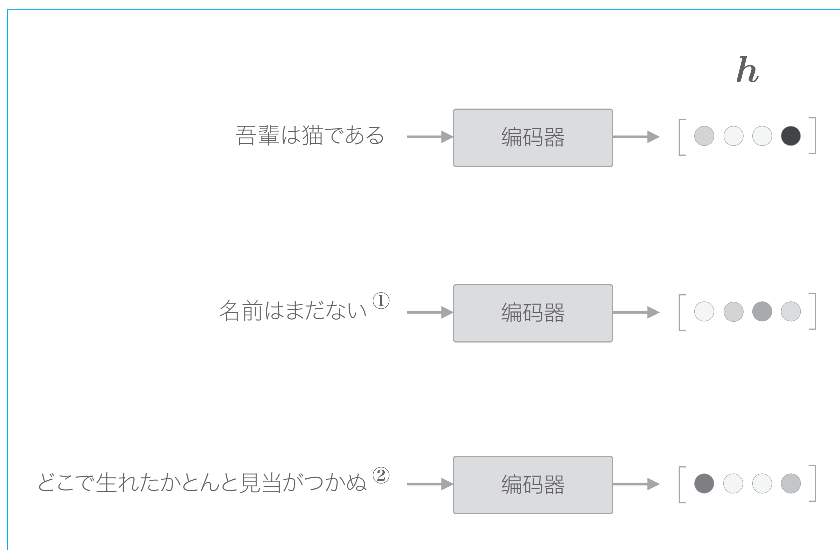


图 7-7　编码器将文本编码为固定长度的向量

如图 7-7 所示，编码器将文本转换为固定长度的向量。那么，解码器是如何“处理”这个编码好的向量，从而生成目标文本的呢？其实，我们已经知道答案了。因为我们只需要直接使用上一节讨论的进行文本生成的模型即可，如图 7-8 所示。

① 《我是猫》中的一句话，译为“还没有名字”。——编者注

② 《我是猫》中的一句话，译为“不记得是在哪里出生的”。——编者注

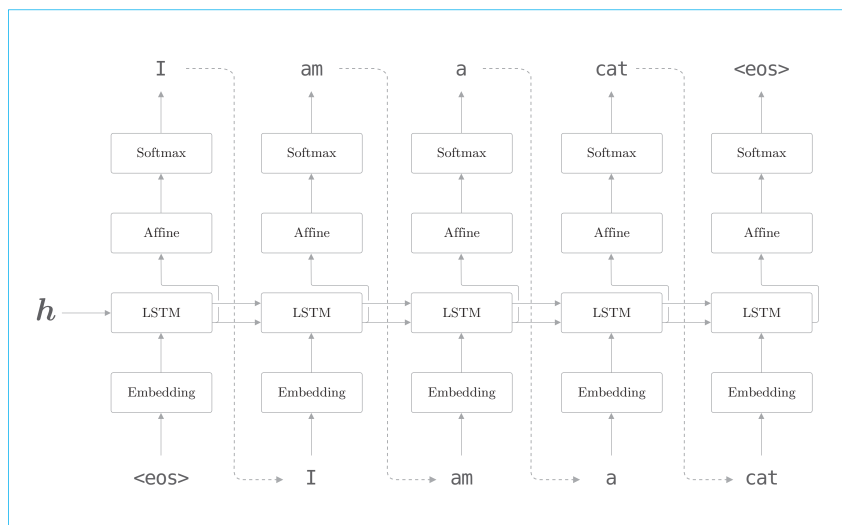


图7-8 解码器的层结构

从图 7-8 中可以看出，解码器的结构和上一节的神经网络完全相同。不过它和上一节的模型存在一点差异，就是 LSTM 层会接收向量 $\boldsymbol{h}$。在上一节的语言模型中，LSTM 层不接收任何信息（硬要说的话，也可以说 LSTM 的隐藏状态接收“0 向量”）。这个唯一的、微小的改变使得普通的语言模型进化为可以驾驭翻译的解码器。



图 7-8 中使用了 <eos> 这一分隔符（特殊符号）。这个分隔符被用作通知解码器开始生成文本的信号。另外，解码器采样到 <eos> 出现为止，所以它也是结束信号。也就是说，分隔符 <eos> 可以用来指示解码器的“开始 / 结束”。在其他文献中，也有使用 <go>、<start> 或者“_”（下划线）作为分隔符的例子。

现在我们连接编码器和解码器，并给出它的层结构，具体如图 7-9 所示。

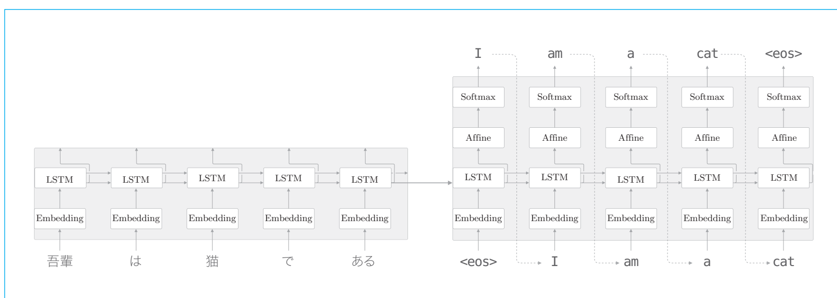


图7-9 seq2seq的整体的层结构

如图 7-9 所示，seq2seq 由两个 LSTM 层构成，即编码器的 LSTM 和解码器的 LSTM。此时，LSTM 层的隐藏状态是编码器和解码器的“桥梁”。在正向传播时，编码器的编码信息通过 LSTM 层的隐藏状态传递给解码器；在反向传播时，解码器的梯度通过这个“桥梁”传递给编码器。

7.2.2 时序数据转换的简单尝试

下面我们来实现 seq2seq，不过在此之前，首先说明一下我们要处理的问题。这里我们将“加法”视为一个时序转换问题。具体来说，如图 7-10 所示，在 seq2seq 学习后，如果输入字符串“57 + 5”，seq2seq 要能正确回答“62”。顺便说一下，这种为了评价机器学习而创建的简单问题，称为“toy problem”。

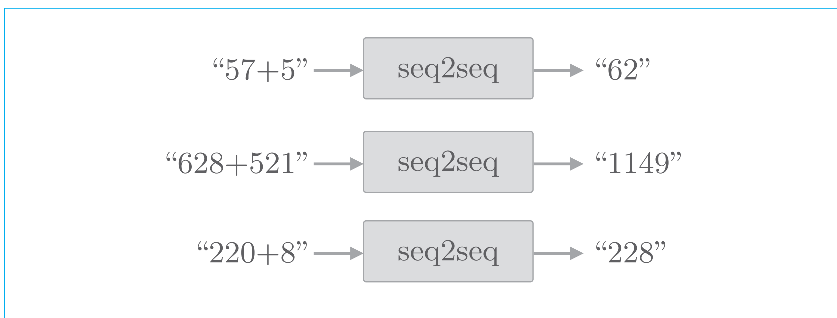


图7-10 让seq2seq学习加法的例子

在我们看来，这里做的加法运算是非常简单的问题，但是 seq2seq 对加法（更确切地说是加法的逻辑）一无所知。seq2seq 从加法的例子（样本）中学习出现的字符模式，这样真的可以学习到加法运算的规则吗？这正是本次实验的看头。

顺便说一下，在之前的 word2vec 和语言模型中，我们都把文本以单词为单位进行了分割，但并非必须这样做。对于本节的这个问题，我们将不以单词为单位，而是以字符为单位进行分割。在以字符为单位进行分割的情况下，“57 + 5”这样的输入会被处理为 `['5', '7', '+', '5']` 这样的列表。

7.2.3 可变长度的时序数据

我们将“加法”视为字符（数字）列表。这里需要注意的是，不同的加法问题（“57 + 5”或者“628 + 521”等）及其回答（“62”或者“1149”等）的字符数是不同的。比如，“57 + 5”共有 4 个字符，而“628 + 521”共有 7 个字符。

如此，在加法问题中，每个样本在时间方向上的大小不同。也就是说，加法问题处理的是可变长度的时序数据。因此，在神经网络的学习中，在进行 mini-batch 处理时，需要想一些应对办法。



在使用批数据进行学习时，会一起处理多个样本。此时，（在我们的实现中）需要保证一个批次内各个样本的数据形状是一致的。

在基于 mini-batch 学习可变长度的时序数据时，最简单的方法是使用填充（padding）。所谓填充，就是用无效（无意义）数据填入原始数据，从而使数据长度对齐。就上面这个加法的例子来说，如图 7-11 所示，在多余位置插入无效字符（这里是空白字符），从而使所有输入数据的长度对齐。

输入

5	7	+	5			
6	2	8	+	5	2	1
2	2	0	+	8		

输出

_	6	2		
_	1	1	4	9
_	2	2	8	

图7-11　为了进行mini-batch学习，使用空白字符进行填充，使输入和输出的大小对齐

本次的问题处理的是 0 ~ 999 的两个数的加法。因此，包括“+”在内，输入的最大字符数是 7。另外，加法的结果最大是 4 个字符（最大为“999 + 999 = 1998”）。因此，对监督数据也进行类似的填充，从而对齐所有样本数据的长度。另外，在本次的问题中，在输出的开始处加上了分隔符“_”（下划线），使得输出数据的字符数统一为 5。这个分隔符作为通知解码器开始生成文本的信号使用。



对于解码器的输出，可以在监督标签中插入表示字符输出结束的分隔符（比如“_62_”或“_1149_”）。但是，简单起见，这里我们不使用表示字符输出结束的分隔符。也就是说，在解码器生成字符串时，始终输出固定数量的字符（这里是包括开始处的“_”在内的5个字符）。

像这样，通过填充对齐数据的大小，可以处理可变长度的时序数据。但是，因为使用了填充，seq2seq 需要处理原本不存在的填充用字符，所以如果追求严谨，使用填充时需要向 seq2seq 添加一些填充专用的处理。比如，在解码器中输入填充时，不应计算其损失（这可以通过向 Softmax with Loss 层添加 mask 功能来解决）。再比如，在编码器中输入填充时，LSTM 层应按原样输出上一时刻的输入。这样一来，LSTM 层就可以像不存在填充一样对输入数据进行编码。

这里的内容有一些复杂，大家即使无法理解也没有关系。为了便于理解，本章我们将填充用字符（空白字符）作为普通数据处理，不进行特别处理。

7.2.4　加法数据集

这里介绍的加法的学习数据预先存放在了 `dataset/addition.txt` 中。如图 7-12 所示，这个文本文件中含有 50 000 个加法样本。这份学习数据的制作参考了 Keras 的 seq2seq 的实现 [40]。

```
 1 16+75  _91
 2 52+607 _659
 3 75+22  _97
 4 63+22  _85
 5 795+3  _798
 6 706+796_1502
 7 8+4    _12
 8 84+317 _401
 9 9+3    _12
10 6+2    _8
11 18+8   _26
12 85+52  _137
13 9+1    _10
14 8+20   _28
15 5+3    _8
Lines: 50,000  Chars: 650,000   650 KB
```

图7-12　加法的学习数据：空白字符（空格）用灰色的点表示

为了使用 Python 轻松处理 seq2seq 的学习数据（文本文件），本书提供了一个专用模块（`dataset/sequence.py`），这个模块有 `load_data()` 和 `get_vocab()` 两个方法。

`load_data(file_name, seed)` 读入由 `file_name` 指定的文本文件，并将文本转换为字符 ID，返回训练数据和测试数据。该方法内部设有随机数种子 `seed` 以打乱数据，分割训练数据和测试数据。另外，`get_vocab()` 方法返回字符与 ID 的映射字典（实际上返回 `char_to_id` 和 `id_to_char`）。现在我们来看一下实际的使用示例（✏ `ch07/show_addition_dataset.py`）。

```
import sys
sys.path.append('..')
from dataset import sequence
```

```
(x_train, t_train), (x_test, t_test) = \
    sequence.load_data('addition.txt', seed=1984)
char_to_id, id_to_char = sequence.get_vocab()

print(x_train.shape, t_train.shape)
print(x_test.shape, t_test.shape)
# (45000, 7) (45000, 5)
# (5000, 7) (5000, 5)

print(x_train[0])
print(t_train[0])
# [ 3  0  2  0  0 11  5]
# [ 6  0 11  7  5]

print(''.join([id_to_char[c] for c in x_train[0]]))
print(''.join([id_to_char[c] for c in t_train[0]]))
# 71+118
# _189
```

像这样，使用 sequence 模块，可以轻松地读入 seq2seq 用的数据。这里，x_train 和 t_train 存放的是字符 ID。另外，字符 ID 和字符之间的映射可以使用 char_to_id 和 id_to_char。



数据集原本应分成训练用、验证用和测试用 3 份。用训练数据进行学习，用验证数据进行调参，最后再用测试数据评价模型的能力。而简单起见，这里只分成训练数据和测试数据 2 份，用它们进行模型的训练和评价。

7.3 seq2seq 的实现

seq2seq 是组合了两个 RNN 的神经网络。这里我们首先将这两个 RNN 实现为 Encoder 类和 Decoder 类，然后将这两个类组合起来，来实现 seq2seq 类。我们先从 Encoder 类开始介绍。

7.3.1 Encoder类

如图 7-13 所示，Encoder 类接收字符串，将其转化为向量 $\boldsymbol{h}$。

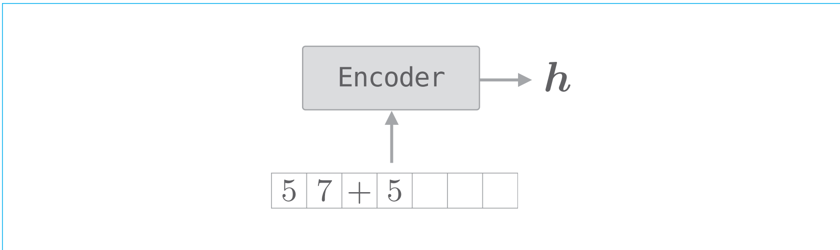


图7-13　Encoder类的输入输出

如前所述，我们使用 RNN 实现编码器。这里，使用 LSTM 层实现图 7-14 的层结构。

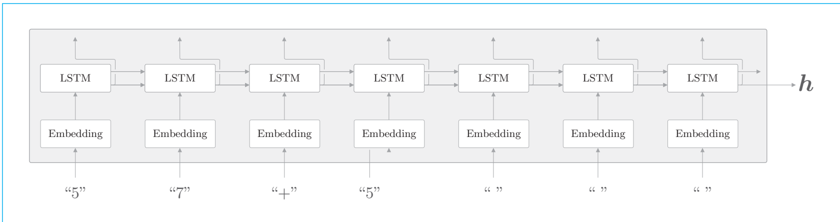
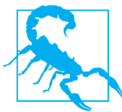


图7-14　编码器的层结构

如图 7-14 所示，`Encoder` 类由 Embedding 层和 LSTM 层组成。Embedding 层将字符（字符 ID）转化为字符向量，然后将字符向量输入 LSTM 层。

LSTM 层向右（时间方向）输出隐藏状态和记忆单元，向上输出隐藏状态。这里，因为上方不存在层，所以丢弃 LSTM 层向上的输出。如图 7-14 所示，在编码器处理完最后一个字符后，输出 LSTM 层的隐藏状态 $\boldsymbol{h}$。然后，这个隐藏状态 $\boldsymbol{h}$ 被传递给解码器。



编码器只将LSTM的隐藏状态传递给解码器。尽管也可以把LSTM的记忆单元传递给解码器，但我们通常不太会把LSTM的记忆单元传递给其他层。这是因为，LSTM的记忆单元被设计为只给自身使用。

顺便说一下，我们已经将时间方向上进行整体处理的层实现为了 Time

LSTM 层和 Time Embedding 层。使用这些 Time 层，我们的编码器将如图 7-15 所示。

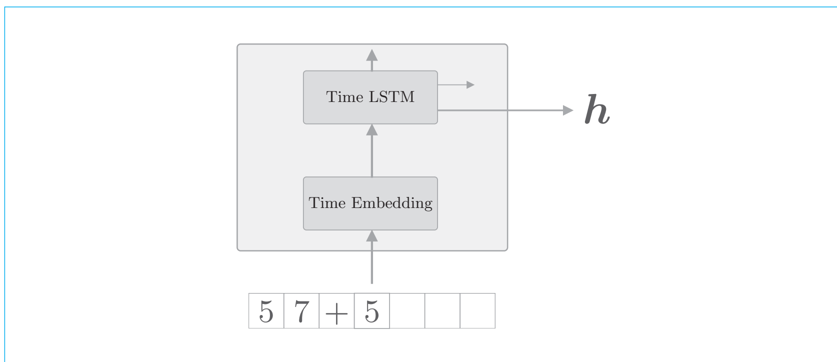


图 7-15　使用 Time 层实现编码器

Encoder 类如下所示。这个 Encoder 类有初始化 __init__()、正向传播 forward() 和反向传播 backward() 这 3 个方法。首先，我们来看一下初始化方法（✏ ch07/seq2seq.py）。

```
class Encoder:
    def __init__(self, vocab_size, wordvec_size, hidden_size):
        V, D, H = vocab_size, wordvec_size, hidden_size
        rn = np.random.randn

        embed_W = (rn(V, D) / 100).astype('f')
        lstm_Wx = (rn(D, 4 * H) / np.sqrt(D)).astype('f')
        lstm_Wh = (rn(H, 4 * H) / np.sqrt(H)).astype('f')
        lstm_b = np.zeros(4 * H).astype('f')

        self.embed = TimeEmbedding(embed_W)
        self.lstm = TimeLSTM(lstm_Wx, lstm_Wh, lstm_b, stateful=False)

        self.params = self.embed.params + self.lstm.params
        self.grads = self.embed.grads + self.lstm.grads
        self.hs = None
```

初始化方法接收 vocab_size、wordvec_size 和 hidden_size 这 3 个参数。vocab_size 是词汇量，相当于字符的种类。顺便说一下，这次共有 13 种字符（数字 0 ~ 9、“+”、“ ”（空白字符）、“_”）。此外，wordvec_size 对应于字符向量的维数，hidden_size 对应于 LSTM 层的隐藏状态的维数。

这个初始化方法进行权重参数的初始化和层的生成。最后，将权重参数和梯度分别归纳在成员变量 params 和 grads 的列表中。因为这次并不保持 Time LSTM 层的状态，所以设定 stateful=False。



第 5 章和第 6 章的语言模型处理的是只有一个长时序数据的问题。那时我们设定 Time LSTM 层的参数 stateful=True，以在保持隐藏状态的同时，处理长时序数据。而这次是有多个短时序数据的问题。因此，针对每个问题重设 LSTM 的隐藏状态（为 0 向量）。

接着来看 forward() 方法和 backward() 方法（✏ ch07/seq2seq.py）。

```
def forward(self, xs):
    xs = self.embed.forward(xs)
    hs = self.lstm.forward(xs)
    self.hs = hs
    return hs[:, -1, :]

def backward(self, dh):
    dhs = np.zeros_like(self.hs)
    dhs[:, -1, :] = dh

    dout = self.lstm.backward(dhs)
    dout = self.embed.backward(dout)
    return dout
```

编码器的正向传播调用 Time Embedding 层和 Time LSTM 层的 forward() 方法，然后取出 Time LSTM 层的最后一个时刻的隐藏状态，将它作为编码器的 forward() 方法的输出。

在编码器的反向传播中，LSTM 层的最后一个隐藏状态的梯度是 dh，

这个 dh 是从解码器传来的梯度。在反向传播的实现中，先生成元素为 0 的张量 dhs，再将 dh 存放到这个 dhs 中的对应位置。剩下的就是调用 Time Embedding 层和 Time LSTM 层的 backward() 方法。以上就是 Encoder 类的实现。

7.3.2　Decoder类

下面，我们继续 Decoder 类的实现。如图 7-16 所示，Decoder 类接收 Encoder 类输出的 $\boldsymbol{h}$，输出目标字符串。

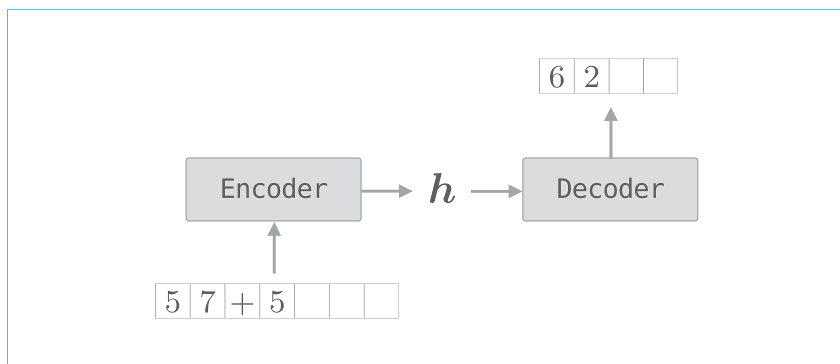


图 7-16　编码器和解码器

如前所述，解码器可以由 RNN 实现。和编码器一样，这里也使用 LSTM 层，此时解码器的层结构如图 7-17 所示。

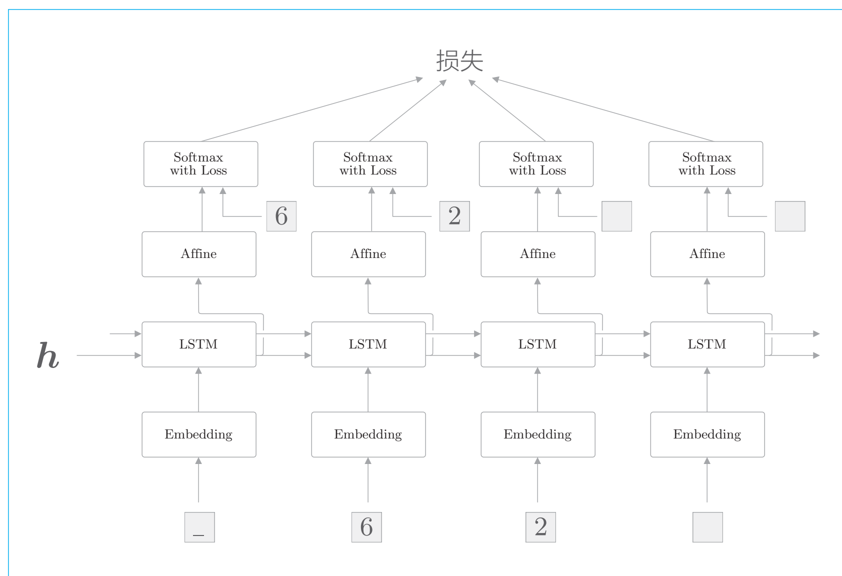


图7-17　解码器的层结构（学习时）

图 7-17 显示了解码器在学习时的层结构。这里使用了监督数据 _62 进行学习，此时输入数据是 `['_', '6', '2', ' ']`，对应的输出是 `['6', '2', ' ', ' ']`。



在使用RNN进行文本生成时，学习时和生成时的数据输入方法不同。在学习时，因为已经知道正确解，所以可以整体地输入时序方向上的数据。相对地，在推理时（生成新字符串时），则只能输入第1个通知开始的分隔符（本次为“_”）。然后，基于输出采样1个字符，并将这个采样出来的字符作为下一个输入，如此重复该过程。

顺便说一句，在 7.1 节，在进行文本生成时，我们基于 Softmax 函数的概率分布进行了采样，因此生成的文本会随机变动。因为这次的问题是加法，所以我们想消除这种概率性的“波动”，生成“确定性的”答案。为此，这次我们仅选择得分最高的字符。也就是说，是“确定性”地选择，而不是“概率性”地选择。图 7-18 显示了解码器生成字符串的过程。

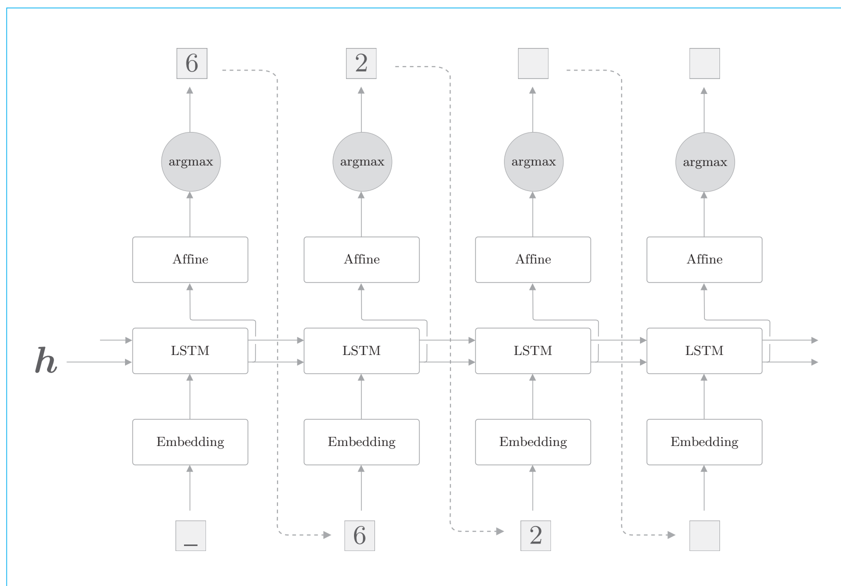


图 7-18　解码器生成字符串的步骤：通过 argmax 节点从 Affine 层的输出中选择最大值的索引（字符 ID）

如图 7-18 所示，这里出现了新的 argmax 节点，这是获取最大值的索引（本例中是字符 ID）的节点。图 7-18 的结构和上一节展示的文本生成时的结构相同。不过这次没有使用 Softmax 层，而是从 Affine 层输出的得分中选择了最大值的字符 ID。



Softmax 层对输入的向量进行正规化。此时，向量元素的值虽然被改变，但是它们的大小关系没有变化。因此，在图 7-18 的情况下，可以省略 Softmax 层。

如上所述，在解码器中，在学习时和在生成时处理 Softmax 层的方式是不一样的。因此，Softmax with Loss 层交给此后实现的 `Seq2seq` 类处理。如图 7-19 所示，`Decoder` 类仅承担 Time Softmax with Loss 层之前的部分。

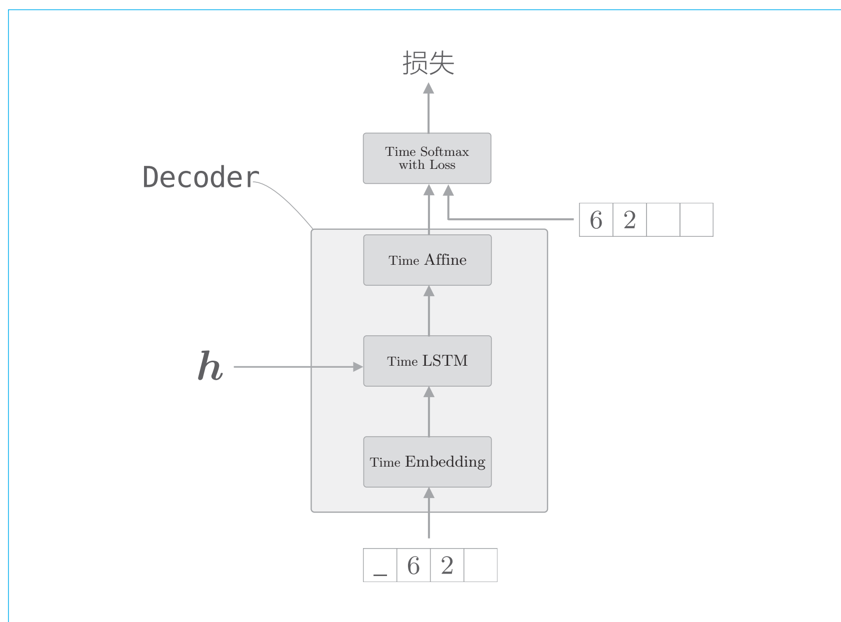


图7-19　Decoder类的结构

由图 7-19 可以看出，Decoder 类由 Time Embedding、Time LSTM 和 Time Affine 这 3 个层构成。下面，我们来看一下 Decoder 层的实现。这里一起给出它的初始化 `__init__()` 方法、正向传播 `forward()` 方法和反向传播 `backward()` 方法（✏ ch07/seq2seq.py）。

```
class Decoder:
    def __init__(self, vocab_size, wordvec_size, hidden_size):
        V, D, H = vocab_size, wordvec_size, hidden_size
        rn = np.random.randn

        embed_W = (rn(V, D) / 100).astype('f')
        lstm_Wx = (rn(D, 4 * H) / np.sqrt(D)).astype('f')
        lstm_Wh = (rn(H, 4 * H) / np.sqrt(H)).astype('f')
        lstm_b = np.zeros(4 * H).astype('f')
        affine_W = (rn(H, V) / np.sqrt(H)).astype('f')
        affine_b = np.zeros(V).astype('f')
```

```
        self.embed = TimeEmbedding(embed_W)
        self.lstm = TimeLSTM(lstm_Wx, lstm_Wh, lstm_b, stateful=True)
        self.affine = TimeAffine(affine_W, affine_b)

        self.params, self.grads = [], []
        for layer in (self.embed, self.lstm, self.affine):
            self.params += layer.params
            self.grads += layer.grads

    def forward(self, xs, h):
        self.lstm.set_state(h)

        out = self.embed.forward(xs)
        out = self.lstm.forward(out)
        score = self.affine.forward(out)
        return score

    def backward(self, dscore):
        dout = self.affine.backward(dscore)
        dout = self.lstm.backward(dout)
        dout = self.embed.backward(dout)
        dh = self.lstm.dh
        return dh
```

这里仅对反向传播进行一些补充说明。在 `backward()` 的实现中，从上方的 Softmax with Loss 层接收梯度 `dscore`，然后按 Time Affine 层、Time LSTM 层和 Time Embedding 层的顺序传播梯度。Time LSTM 层将时间方向上的梯度保存在 Time LSTM 层的成员变量 `dh` 中（具体请参考 6.3 节），因此取出时间方向上的梯度 `dh`，将其作为 `Decoder` 类的 `backward()` 的输出。

如前所述，`Decoder` 类在学习时和在生成文本时的行为不同。上面的 `forward()` 方法是假定在学习时使用的。我们将 `Decoder` 类生成文本时的方法实现为 `generate()`。

```
    def generate(self, h, start_id, sample_size):
        sampled = []
        sample_id = start_id
        self.lstm.set_state(h)
```

```
        for _ in range(sample_size):
            x = np.array(sample_id).reshape((1, 1))
            out = self.embed.forward(x)
            out = self.lstm.forward(out)
            score = self.affine.forward(out)

            sample_id = np.argmax(score.flatten())
            sampled.append(int(sample_id))

        return sampled
```

这个 generate() 方法有 3 个参数，分别是从编码器接收的隐藏状态 h、最开始输入的字符 ID start_id 和生成的字符数量 sample_size。这里重复如下操作：输入一个字符，选择 Affine 层输出的得分中最大值的字符 ID。以上就是 Decoder 类的实现。



在这次的问题中，需要将编码器的输出 h 设定给解码器的 Time LSTM 层。此时，通过设置 Time LSTM 层为 stateful，可以不重设隐藏状态，在保持编码器的 h 的同时，进行正向传播。

7.3.3　Seq2seq 类

最后来看 Seq2seq 类的实现。话虽如此，这里需要做的只是将 Encoder 类和 Decoder 类连接在一起，然后使用 Time Softmax with Loss 层计算损失而已。Seq2seq 类的实现如下所示（✏ ch07/seq2seq.py）。

```
class Seq2seq(BaseModel):
    def __init__(self, vocab_size, wordvec_size, hidden_size):
        V, D, H = vocab_size, wordvec_size, hidden_size
        self.encoder = Encoder(V, D, H)
        self.decoder = Decoder(V, D, H)
        self.softmax = TimeSoftmaxWithLoss()

        self.params = self.encoder.params + self.decoder.params
        self.grads = self.encoder.grads + self.decoder.grads
```

```
    def forward(self, xs, ts):
        decoder_xs, decoder_ts = ts[:, :-1], ts[:, 1:]

        h = self.encoder.forward(xs)
        score = self.decoder.forward(decoder_xs, h)
        loss = self.softmax.forward(score, decoder_ts)
        return loss

    def backward(self, dout=1):
        dout = self.softmax.backward(dout)
        dh = self.decoder.backward(dout)
        dout = self.encoder.backward(dh)
        return dout

    def generate(self, xs, start_id, sample_size):
        h = self.encoder.forward(xs)
        sampled = self.decoder.generate(h, start_id, sample_size)
        return sampled
```

Encoder 和 Decoder 的各类中已经实现了主要的处理，因此这里只是将它们组合起来。以上就是 Seq2seq 类的实现。下面我们使用这个 Seq2seq 类，来挑战一下加法问题。

7.3.4 seq2seq的评价

Seq2seq 的学习和基础神经网络的学习具有相同的流程。基础神经网络的学习流程如下：

1. 从训练数据中选择一个 mini-batch
2. 基于 mini-batch 计算梯度
3. 使用梯度更新权重

这里使用 1.4.4 节说明过的 Trainer 类进行上述操作。另外，seq2seq 针对每个 epoch 求解测试数据（生成字符串），并计算正确率。seq2seq 的学习代码如下所示（✏ ch07/train_seq2seq.py）。

```
import sys
sys.path.append('..')
import numpy as np
import matplotlib.pyplot as plt
from dataset import sequence
from common.optimizer import Adam
from common.trainer import Trainer
from common.util import eval_seq2seq
from seq2seq import Seq2seq
from peeky_seq2seq import PeekySeq2seq


# 读入数据集
(x_train, t_train), (x_test, t_test) = sequence.load_data('addition.txt')
char_to_id, id_to_char = sequence.get_vocab()

# 设定超参数
vocab_size = len(char_to_id)
wordvec_size = 16
hidden_size = 128
batch_size = 128
max_epoch = 25
max_grad = 5.0

# 生成模型/优化器/训练器
model = Seq2seq(vocab_size, wordvec_size, hidden_size)
optimizer = Adam()
trainer = Trainer(model, optimizer)

acc_list = []
for epoch in range(max_epoch):
    trainer.fit(x_train, t_train, max_epoch=1,
                batch_size=batch_size, max_grad=max_grad)

    correct_num = 0
    for i in range(len(x_test)):
        question, correct = x_test[[i]], t_test[[i]]
        verbose = i < 10
        correct_num += eval_seq2seq(model, question, correct,
                                    id_to_char, verbose)
    acc = float(correct_num) / len(x_test)
    acc_list.append(acc)
    print('val acc %.3f%%' % (acc * 100))
```

这里显示的代码和基础神经网络的学习用代码是一样的，不过这里采用正确率（正确回答了多少问题）作为评价指标。具体来说，就是针对每个epoch 对正确回答了测试数据中的多少问题进行统计。

为了测量上述实现的正确率，我们使用 `common/util.py` 中的 `eval_seq2seq(model, question, correct, id_to_char, verbose, is_reverse)` 方法。这个方法向模型输入问题，生成字符串，并判断它是否与答案相符。如果模型给出的答案正确，则返回 1；如果错误，则返回 0.



`eval_seq2seq(model, question, correct, id_to_char, verbose, is_reverse)` 方法有 6 个参数。首先是模型 `model`、问题（字符 ID 数组）`question`、正确解（字符 ID 列表）`correct`。之后是进行字符 ID 与字符映射的字典 `id_to_char`、指定是否显示结果的 `verbose`、指定是否反转输入语句的 `is_reverse`。如果设置 `verbose = True`，则结果会显示在终端上。这次的实验仅显示最初的 10 份测试数据。另外，关于参数 `is_reverse`，我们稍后再解释。

运行上述代码后，图 7-20 所示的结果会显示在终端（控制台）上 [①]。

```
ch07 — -bash — 50×17
---
Q 600+257
T 857
☒ 864
---
Q 761+292
T 1053
☒ 1059
---
Q 830+597
T 1427
☒ 1444
---
Q 26+838
T 864
☑ 864
---
```

图 7-20　终端上显示的部分结果

① 在Windows环境中，☑☒的显示可能有所不同。

如图 7-20 所示，在终端上，每个 epoch 显示一次结果。每行有“Q 600 + 257”这样的问题语句，它的下方是“T 857”这样的正确答案。这里，“☒ 864”是我们的模型给出的答案。如果我们的模型给出了正确答案，则终端上会显示“☑ 864”。

我们再看一下随着学习的进行，上面的结果会发生什么样的变化。图 7-21 给出了一个例子。

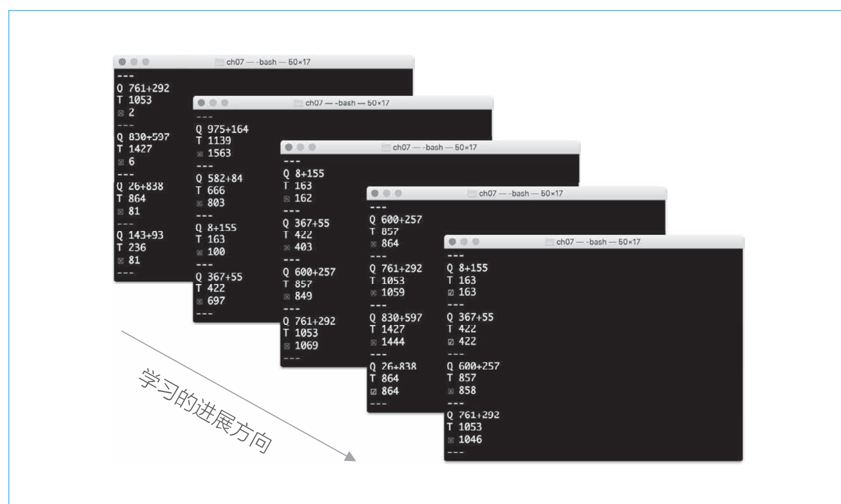


图 7-21 终端上显示的结果的变化

图 7-21 中展示了随着学习的进行而出现的几个结果。从结果中可知，seq2seq 最开始没能顺利回答问题。但是，随着学习不断进行，它在慢慢靠近正确答案，然后变得可以正确回答一些问题。下面，我们来绘制一下每个 epoch 的正确率，结果如图 7-22 所示。

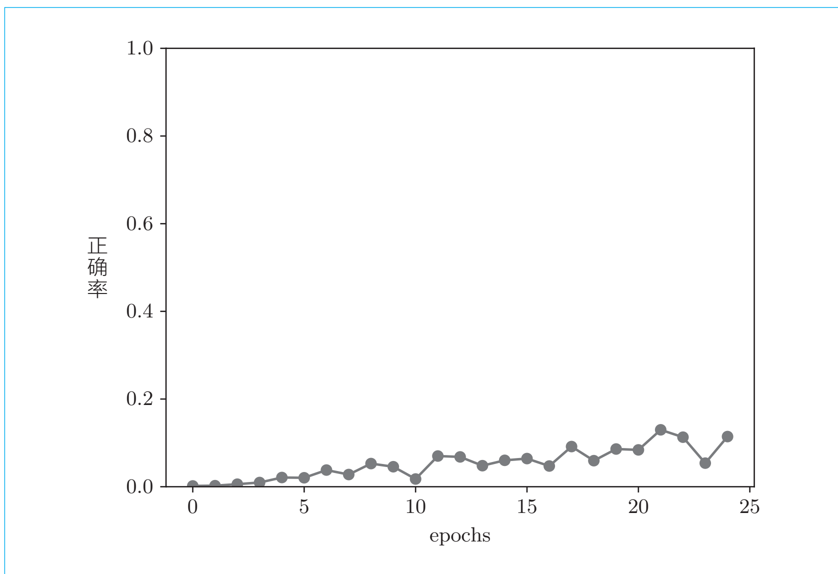


图 7-22 正确率的变化

由图 7-22 可知，随着学习的积累，正确率稳步提高。本次的实验只进行了 25 次，最后的正确率约为 10%。从图中的变化趋势可知，如果继续学习，正确率应该还会进一步上升。不过，为了能更好地学习相同的问题（加法问题），这里我们暂停本次学习，对 seq2seq 进行一些改进。

7.4 seq2seq 的改进

本节我们对上一节的 seq2seq 进行改进，以改进学习的进展。为了达成该目标，可以使用一些比较有前景的技术。本节我们展示其中的两个方案，并基于实验确认它们的效果。

7.4.1 反转输入数据（Reverse）

第一个改进方案是非常简单的技巧。如图 7-23 所示，反转输入数据的顺序。

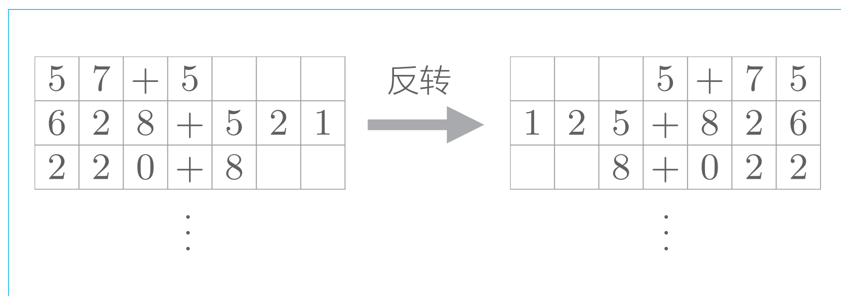


图7-23　反转输入数据的例子

这个反转输入数据的技巧是文献 [41] 中提出来的。据研究，在许多情况下，使用这个技巧后，学习进展得更快，最终的精度也有提高。现在我们来做一下实验。

为了反转输入数据，对于上一节的学习用代码（ch07/train_seq2seq.py），在读入数据集之后，我们追加下面的代码。

```
# 读入数据集
(x_train, t_train), (x_test, t_test) = sequence.load_data('addition.txt')
...
x_train, x_test = x_train[:, ::-1], x_test[:, ::-1]
...
```

如上所示，可以使用 x_train[:, ::-1] 反转数组的排列。那么，通过反转输入数据，正确率可以上升多少呢？结果如图 7-24 所示。

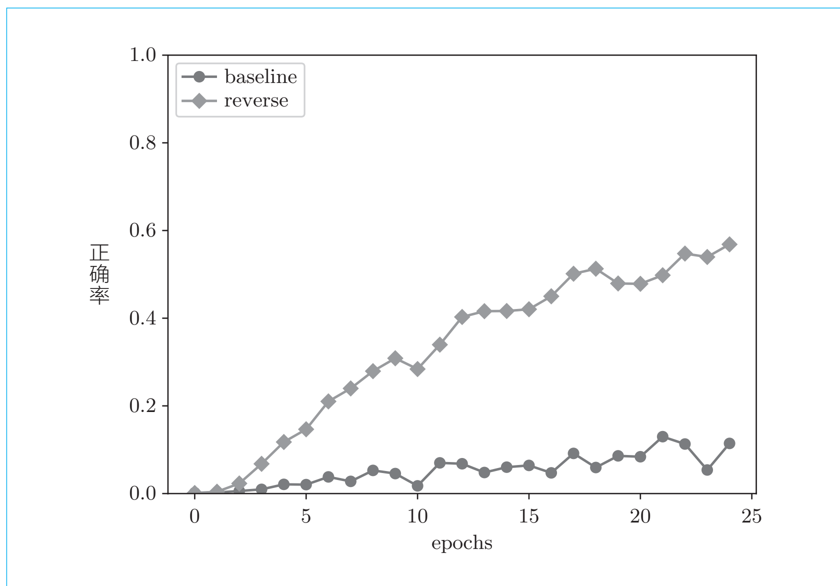


图 7-24 seq2seq 的正确率的变化：baseline 是上一节的结果，reverse 是反转输入数据后的结果

从图 7-24 中可知，仅仅通过反转输入数据，学习的进展就得到了改善！在 25 个 epoch 时，正确率为 50% 左右。再次重复一遍，这里和上一次（图中的 baseline）的差异只是将数据反转了一下。仅仅这样，就产生了这么大的差异，真是令人吃惊。当然，虽然反转数据的效果因任务而异，但是通常都会有好的结果。

为什么反转数据后，学习进展变快，精度提高了呢？虽然理论上不是很清楚，但是直观上可以认为，反转数据后梯度的传播可以更平滑。比如，考虑将“吾輩 は 猫 で ある”① 翻译成“I am a cat”这一问题，单词“吾輩”和单词“I”之间有转换关系。此时，从“吾輩”到“I”的路程必须经过“は”“猫”“で”“ある”这 4 个单词的 LSTM 层。因此，在反向传播时，梯度从“I”抵达“吾輩”，也要受到这个距离的影响。

① 译为“我是猫”，不懂日文的读者只需明白该句可以拆分为“吾輩”“は”“猫”“で”“ある”这 5 个单词即可。——编者注

那么，如果反转输入语句，也就是变为“ある で 猫 は 吾輩”，结果会怎样呢？此时，“吾輩”和“I”彼此相邻，梯度可以直接传递。如此，因为通过反转，输入语句的开始部分和对应的转换后的单词之间的距离变近（这样的情况变多），所以梯度的传播变得更容易，学习效率也更高。不过，在反转输入数据后，单词之间的“平均”距离并不会发生改变。

7.4.2 偷窥（Peeky）

接下来是 seq2seq 的第二个改进。在进入正题之前，我们再看一下编码器的作用。如前所述，编码器将输入语句转换为固定长度的向量 $\boldsymbol{h}$，这个 $\boldsymbol{h}$ 集中了解码器所需的全部信息。也就是说，它是解码器唯一的信息源。但是，如图 7-25 所示，当前的 seq2seq 只有最开始时刻的 LSTM 层利用了 $\boldsymbol{h}$。我们能更加充分地利用这个 $\boldsymbol{h}$ 吗？

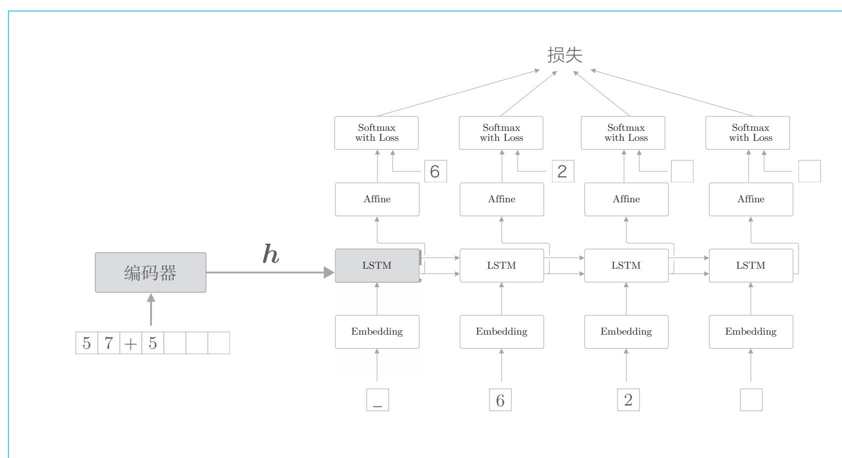


图7-25 改进前：只有最开始的LSTM层接收编码器的输出 $\boldsymbol{h}$

为了达成该目标，seq2seq 的第二个改进方案就应运而生了。具体来说，就是将这个集中了重要信息的编码器的输出 $\boldsymbol{h}$ 分配给解码器的其他层。我们的解码器可以考虑图 7-26 中的网络结构。

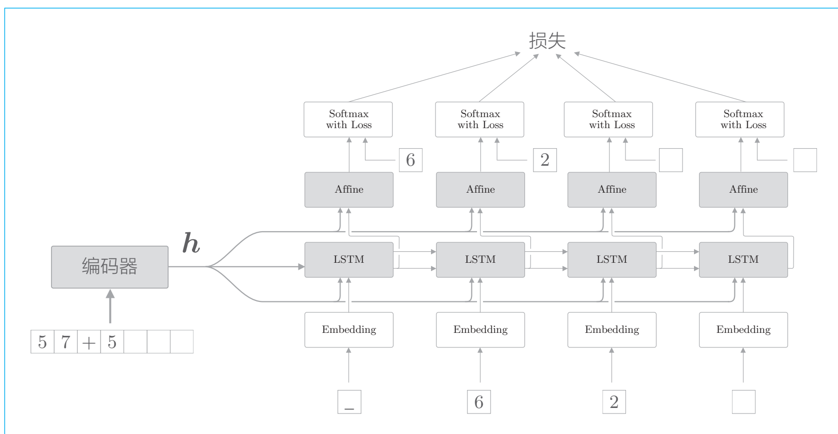


图 7-26　改进后：将编码器的输出 h 分配给所有时刻的 LSTM 层和 Affine 层

如图 7-26 所示，将编码器的输出 h 分配给所有时刻的 Affine 层和 LSTM 层。比较图 7-26 和图 7-25 可知，之前 LSTM 层专用的重要信息 h 现在在多个层（在这个例子中有 8 个层）中共享了。重要的信息不是一个人专有，而是多人共享，这样我们或许可以做出更加正确的判断。



这里的改进是将编码好的信息分配给解码器的其他层，这可以解释为其他层也能“偷窥”到编码信息。因为“偷窥”的英语是 peek，所以将这个改进了的解码器称为 Peeky Decoder。同理，将使用了 Peeky Decoder 的 seq2seq 称为 Peeky seq2seq。这个想法基于文献 [42]。

在图 7-26 中，有两个向量同时被输入到了 LSTM 层和 Affine 层，这实际上表示两个向量的拼接（concatenate）。因此，在刚才的图中，如果使用 concat 节点拼接两个向量，则正确的计算图可以绘制成图 7-27。

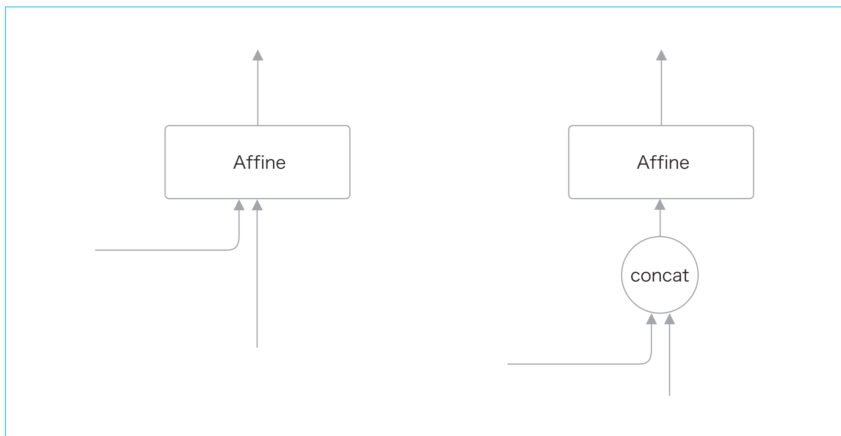


图7-27 在Affine层的输入有两个的情况下（左图），将它们拼接起来输入Affine层（右图）

下面给出 PeekyDecoder 类的实现。这里仅显示初始化 __init__() 方法和正向传播 forward() 方法。因为没有特别难的地方，所以这里省略了反向传播 backward() 方法和文本生成 generate() 方法（✏ ch07/peeky_seq2seq.py）。

```
class PeekyDecoder:
    def __init__(self, vocab_size, wordvec_size, hidden_size):
        V, D, H = vocab_size, wordvec_size, hidden_size
        rn = np.random.randn

        embed_W = (rn(V, D) / 100).astype('f')
        lstm_Wx = (rn( H + D , 4 * H) / np.sqrt(H + D)).astype('f')
        lstm_Wh = (rn(H, 4 * H) / np.sqrt(H)).astype('f')
        lstm_b = np.zeros(4 * H).astype('f')
        affine_W = (rn( H + H , V) / np.sqrt(H + H)).astype('f')
        affine_b = np.zeros(V).astype('f')

        self.embed = TimeEmbedding(embed_W)
        self.lstm = TimeLSTM(lstm_Wx, lstm_Wh, lstm_b, stateful=True)
        self.affine = TimeAffine(affine_W, affine_b)

        self.params, self.grads = [], []
        for layer in (self.embed, self.lstm, self.affine):
            self.params += layer.params
            self.grads += layer.grads
        self.cache = None
```

```
    def forward(self, xs, h):
        N, T = xs.shape
        N, H = h.shape

        self.lstm.set_state(h)

        out = self.embed.forward(xs)
        hs = np.repeat(h, T, axis=0).reshape(N, T, H)
        out = np.concatenate((hs, out), axis=2)

        out = self.lstm.forward(out)
        out = np.concatenate((hs, out), axis=2)

        score = self.affine.forward(out)
        self.cache = H
        return score
```

PeekyDecoder 的初始化和上一节的 Decoder 基本上是一样的，不同之处仅在于 LSTM 层权重和 Affine 层权重的形状。因为这次的实现要接收编码器编码好的向量，所以权重参数的形状相应地变大了。

接着是 forward() 的实现。这里首先使用 np.repeat() 根据时序大小复制相应份数的 h，并将其设置为 hs。然后，将 hs 和 Embedding 层的输出用 np.concatenate() 拼接，并输入 LSTM 层。同样地，Affine 层的输入也是 hs 和 LSTM 层的输出的拼接。



编码器和上一节没有变化，因此直接使用上一节的编码器。

最后，我们来实现 PeekySeq2seq，不过这和上一节的 Seq2seq 类基本相同，唯一的区别是 Decoder 层。上一节的 Seq2seq 类使用了 Decoder 类，与此相对，这里使用 PeekyDecoder，剩余的逻辑完全一样。因此，PeekySeq2seq 类的实现只需要继承上一章的 Seq2seq 类，并修改一下初始化部分（✏ ch07/peeky_seq2seq.py）。

```
from seq2seq import Seq2seq, Encoder


class PeekySeq2seq(Seq2seq):
    def __init__(self, vocab_size, wordvec_size, hidden_size):
        V, D, H = vocab_size, wordvec_size, hidden_size
        self.encoder = Encoder(V, D, H)
        self.decoder = PeekyDecoder(V, D, H)
        self.softmax = TimeSoftmaxWithLoss()

        self.params = self.encoder.params + self.decoder.params
        self.grads = self.encoder.grads + self.decoder.grads
```

至此，准备工作就完成了。现在我们使用这个 PeekySeq2seq 类，再次挑战加法问题。学习用代码仍使用上一节的代码，只需要将 Seq2seq 类换成 PeekySeq2seq 类。

```
# model = Seq2seq(vocab_size, wordvec_size, hidden_size)
model = PeekySeq2seq(vocab_size, wordvec_size, hidden_size)
```

这里，我们在第一个改进（反转输入）的基础上进行实验，结果如图 7-28 所示。

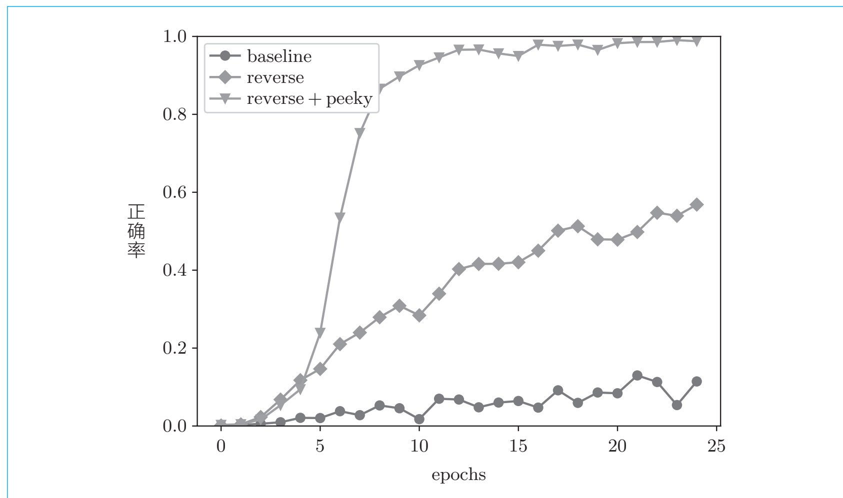


图 7-28　“reverse ＋ peeky”是进行了本节的两个改进的结果

如图 7-28 所示，加上了 Peeky 的 seq2seq 的结果大幅变好。刚过 10 个 epoch 时，正确率已经超过 90%，最终的正确率接近 100%。

从上述实验结果可知，Reverse 和 Peeky 都有很好的效果。借助反转输入语句的 Reverse 和共享编码器信息的 Peeky，我们获得了令人满意的结果！

这样我们就结束了对 seq2seq 的改进，不过故事仍在继续。实际上，本节的改进只能说是“小改进”，下一章我们将对 seq2seq 进行“大改进”。届时将使用名为 Attention 的技术，它能使 seq2seq 发生巨大变化。

这里的实验有几个需要注意的地方。因为使用 Peeky 后，网络的权重参数会额外地增加，计算量也会增加，所以这里的实验结果必须考虑到相应地增加的“负担”。另外，seq2seq 的精度会随着超参数的调整而大幅变化。虽然这里的结果是可靠的，但是在实际问题中，它的效果可能不稳定。

7.5 seq2seq的应用

seq2seq 将某个时序数据转换为另一个时序数据，这个转换时序数据的框架可以应用在各种各样的任务中，比如以下几个例子。

- 机器翻译：将“一种语言的文本”转换为“另一种语言的文本”
- 自动摘要：将“一个长文本”转换为“短摘要”
- 问答系统：将“问题”转换为“答案”
- 邮件自动回复：将“接收到的邮件文本”转换为“回复文本”

像这样，seq2seq 可以用于处理成对的时序数据的问题。除了自然语言之外，也可以用于语音、视频等数据。有些乍一看不属于 seq2seq 的问题，通过对输入输出数据进行预处理，也可以应用 seq2seq。本节将介绍几个使用 seq2seq 的应用。如果读者能由此感受到 seq2seq 的潜力和乐趣，那就再好不过了。

7.5.1 聊天机器人

聊天机器人是人和计算机使用文本进行对话的程序，现在已经被用于Facebook、Twitter 等各种各样的服务中。

毫无疑问，聊天机器人可以使用 seq2seq。因为对话是由“对方的发言”和“本方的发言”构成的，可以理解为是将“对方的发言”转换为“本方的发言”的问题。也就是说，如果有对话文本数据，seq2seq 就可以学习它。

另外，聊天机器人也可以用在实用场景中。文献 [43] 进行了将基于 seq2seq 的聊天机器人应用于 IT 帮助台的实验，并展示了它成功解决有关 VPN 接入问题的例子。这里，我们将其中的一部分对话翻译成中文（图 7-29）[①]。

机器：你好。
人：你好。
机器：请问你使用的是什么操作系统？
人：Linux。

……

机器：运行时报的错误是什么？
人：报了一个有关 VPN 的错误。
机器：在用 VPN 联网时，显示了什么错误消息？
人：连接未成功之类的。

……

机器：现在你是在公司还是在家？
人：在家。
机器：那你登录一下这个网址 <URL>，输入你的用户名和密码。
人：好了。
机器：试试可不可以用 <NAME> 正常登录。
人：好像可以了。
机器：好的。还有其他问题吗？
人：没了，太感谢了。

图 7-29 使用基于 seq2seq 的聊天机器人进行对话的例子（参考文献 [43]）

① 原文是英语对话，本书日文原版中作者翻译为了日文，这里译者翻译为了中文。——译者注

从图 7-29 中可知，机器（聊天机器人）很好地解决了问题，遇到 VPN 接入问题的人被引导到了能解决该问题的 URL。当然，它只能解决与 IT 帮助台相关的问题，无法泛化。但是，基于对话获取答案或者线索，这一点非常实用，应用范围很广。实际上，这样的服务（简易版）已经可以在若干网站上看到。

7.5.2　算法学习

本章进行的 seq2seq 实验是加法这样的简单问题，但理论上它也能处理更加高级的问题，比如图 7-30 所示的 Python 代码。

Input:
```
j=8584
for x in range(8):
  j+=920
b=(1500+j)
print((b+7567))
```
Target: 25011.

Input:
```
i=8827
c=(i-5347)
print((c+8704) if 2641<8500 else 5308)
```
Target: 12184.

图 7-30　用 Python 写的代码示例：图中的 Input 是输入，Target 是输出（引自文献 [44]）

源代码也是用字符编写的时序数据。我们可以将跨行的代码处理为一条语句（将换行视为换行符）。因此，可以直接将源代码输入 seq2seq，让 seq2seq 对源代码与目标答案一起进行学习。

上述包含 `for` 语句和 `if` 语句的问题不太容易解决。不过，即便是这样的问题，也可以在 seq2seq 框架内处理。通过改造 seq2seq 的结构，可以期待这样的问题能够被解决。



下一章将介绍 RNN 的扩展——NTM（Neural Turing Machine，神经图灵机）模型。届时计算机（图灵机）将学习内存的读写顺序，重现算法。

7.5.3　自动图像描述

到目前为止，我们只看了处理文本的 seq2seq 的应用示例，除了文本之外，seq2seq 还可以处理图像、语音等类型的数据。本节我们来看一下将图像转换为文本的自动图像描述（image captioning）[45] [46]。

自动图像描述将“图像”转换为“文本”。如图 7-31 所示，这也可以在 seq2seq 的框架下解决。

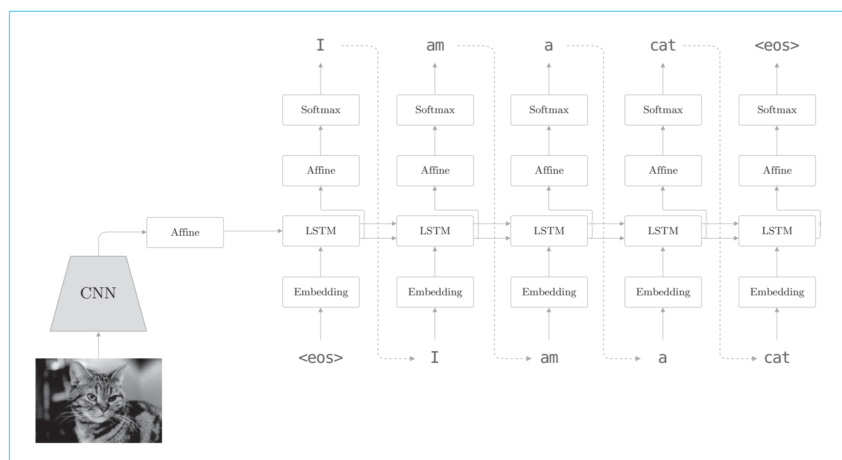


图 7-31　用于自动图像描述的 seq2seq 的网络结构示例

图 7-31 是我们熟悉的网络结构。实际上，它和之前的网络的唯一区别在于，编码器从 LSTM 换成了 CNN（Convolutional Neural Network，卷积神经网络），而解码器仍使用与之前相同的网络。仅通过这点改变（用 CNN 替代 LSTM），seq2seq 就可以处理图像了。

这里补充说明一下图 7-31 中的 CNN。此处，CNN 对图像进行编码，这时 CNN 的最终输出是特征图。因为特征图是三维（高、宽、通道）的，所以需要想一些办法让解码器的 LSTM 可以处理它。于是，我们将 CNN 的特征图扁平化到一维，并基于全连接的 Affine 层进行转换。之后，再将转换后的数据传递给解码器，就可以像之前一样生成文本了。



图 7-31 的 CNN 使用 VGG、ResNet 等成熟网络，并使用在别的图像数据集（ImageNet 等）上学习好的权重，这样可以获得好的编码，从而生成好的文本。

现在我们看几个基于 seq2seq 的自动图像描述的例子。图 7-32 显示的是由基于 TensorFlow 的 im2txt[47] 生成的例子。此处使用的网络基于图 7-31，并在其上进行了若干改进。

A person on a beach flying a kite.
（在海滩放风筝的人）



A black and white photo of a train on a train track.
（铁轨上的一辆火车的黑白照片）



A person skiing down a snow covered slope.
（在积雪的斜坡上滑雪的人）



A group of giraffe standing next to each other.
（一群长颈鹿站在一起）

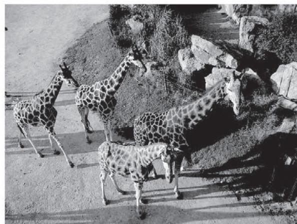


图 7-32　自动图像描述的例子：将图像转换为文本（参考文献 [47]）

由图 7-32 可知，这里得到了很不错的结果。之所以能够达到这样的效果，是因为存在大量的图像和说明文字等训练数据（比如，ImageNet 等大

规模的图像数据）。再加上可以高效学习这些训练数据的 seq2seq 的应用，最终得到了图 7-32 所示的出色结果。

7.6　小结

本章我们探讨了基于 RNN 的文本生成。实际上，我们只是稍微改动了一下上一章的基于 RNN 的语言模型，增加了文本生成的功能。在本章后半部分，我们研究了 seq2seq，并使之成功学习了简单的加法。seq2seq 模型拼接了编码器和解码器，是组合了两个 RNN 的简单结构。但是，尽管 seq2seq 简单，却具有巨大的潜力，可以用于各种各样的应用。

另外，本章还介绍了改进 seq2seq 的两个方案——Reverse 和 Peeky。我们对这两个方案进行了实现和评价，并确认了它们的效果。下一章我们将继续改进 seq2seq，届时深度学习中最重要的技巧之一 Attention 将会出现。我们将说明 Attention 的机制，然后基于它实现更强大的 seq2seq。

本章所学的内容

- 基于 RNN 的语言模型可以生成新的文本
- 在进行文本生成时，重复“输入一个单词（字符），基于模型的输出（概率分布）进行采样”这一过程
- 通过组合两个 RNN，可以将一个时序数据转换为另一个时序数据（seq2seq）
- 在 seq2seq 中，编码器对输入语句进行编码，解码器接收并解码这个编码信息，获得目标输出语句
- 反转输入语句（Reverse）和将编码信息分配给解码器的多个层（Peeky）可以有效提高 seq2seq 的精度
- seq2seq 可以用在机器翻译、聊天机器人和自动图像描述等各种各样的应用中

第8章
Attention

注意力是全部。

——Vaswani 们的论文标题 [52]

上一章我们使用 RNN 生成了文本，又通过连接两个 RNN，将一个时序数据转换为了另一个时序数据。我们将这个网络称为 seq2seq，并用它成功求解了简单的加法问题。之后，我们对这个 seq2seq 进行了几处改进，几乎完美地解决了这个简单的加法问题。

本章我们将进一步探索 seq2seq 的可能性（以及 RNN 的可能性）。这里，Attention 这一强大而优美的技术将登场。Attention 毫无疑问是近年来深度学习领域最重要的技术之一。本章的目标是在代码层面理解 Attention 的结构，然后将其应用于实际问题，体验它的奇妙效果。

8.1　Attention 的结构

如上一章所述，seq2seq 是一个非常强大的框架，应用面很广。这里我们将介绍进一步强化 seq2seq 的注意力机制（attention mechanism，简称 Attention）。基于 Attention 机制，seq2seq 可以像我们人类一样，将“注意力”集中在必要的信息上。此外，使用 Attention 可以解决当前 seq2seq 面临的问题。

本节我们将首先指出当前 seq2seq 存在的问题，然后一边说明 Attention 的结构，一边对其进行实现。



上一章我们已经对seq2seq进行了改进，但那些只能算是“小改进”。下面将要说明的Attention技术才是解决seq2seq的问题的“大改进”。

8.1.1　seq2seq存在的问题

seq2seq中使用编码器对时序数据进行编码，然后将编码信息传递给解码器。此时，编码器的输出是固定长度的向量。实际上，这个“固定长度”存在很大问题。因为固定长度的向量意味着，无论输入语句的长度如何（无论多长），都会被转换为长度相同的向量。以上一章的翻译为例，如图8-1所示，不管输入的文本如何，都需要将其塞入一个固定长度的向量中。

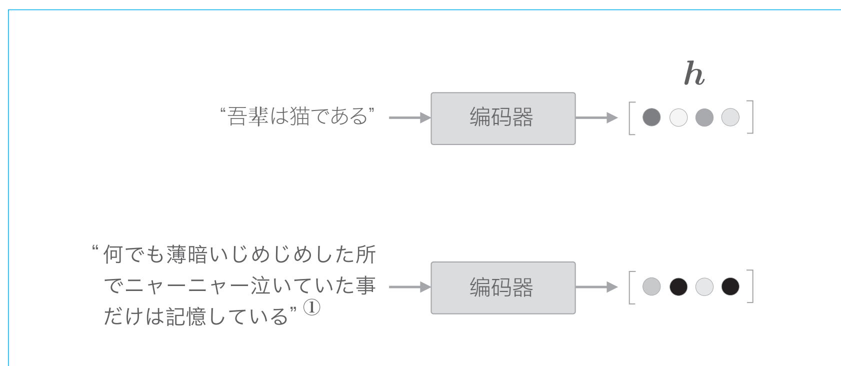


图8-1　无论输入语句多长，编码器都将其塞入固定长度的向量中

无论多长的文本，当前的编码器都会将其转换为固定长度的向量。就像把一大堆西装塞入衣柜里一样，编码器强行把信息塞入固定长度的向量中。但是，这样做早晚会遇到瓶颈。就像最终西服会从衣柜中掉出来一样，有用的信息也会从向量中溢出。

现在我们就来改进seq2seq。首先改进编码器，然后再改进解码器。

①《我是猫》中的一句话，译为“只记得我在一个昏暗潮湿的地方喵喵地哭泣着”。——编者注

8.1.2 编码器的改进

到目前为止，我们都只将 LSTM 层的最后的隐藏状态传递给解码器，但是编码器的输出的长度应该根据输入文本的长度相应地改变。这是编码器的一个可以改进的地方。具体而言，如图 8-2 所示，使用各个时刻的 LSTM 层的隐藏状态。

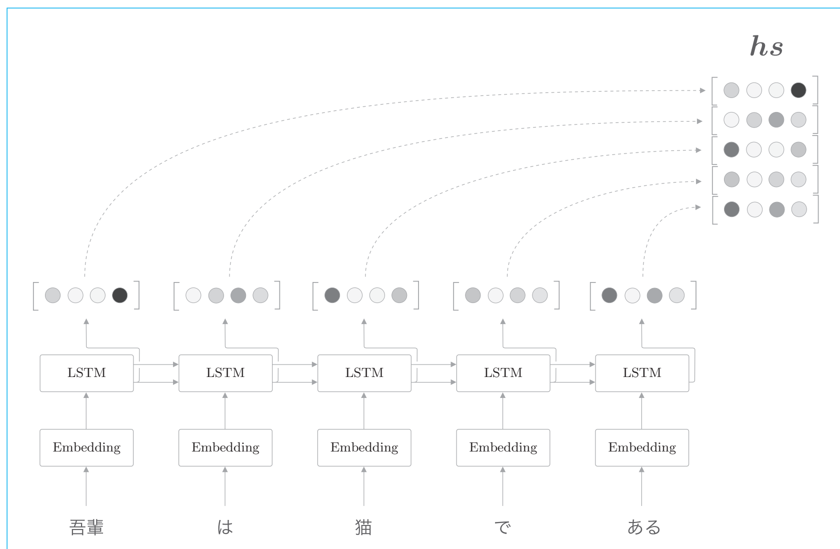


图 8-2 使用编码器各个时刻（各个单词）的 LSTM 层的隐藏状态（这里表示为 hs）

如图 8-2 所示，使用各个时刻（各个单词）的隐藏状态向量，可以获得和输入的单词数相同数量的向量。在图 8-2 的例子中，输入了 5 个单词，此时编码器输出 5 个向量。这样一来，编码器就摆脱了“一个固定长度的向量”的制约。



在许多深度学习框架中，在初始化RNN层(或者LSTM层、GRU层等)时，可以选择是返回“全部时刻的隐藏状态向量”，还是返回“最后时刻的隐藏状态向量”。比如，在Keras中，在初始化RNN层时，可以设置`return_sequences`为`True`或者`False`。

图 8-2 中我们需要关注 LSTM 层的隐藏状态的“内容”。此时，各个时刻的 LSTM 层的隐藏状态都充满了什么信息呢？有一点可以确定的是，各个时刻的隐藏状态中包含了大量当前时刻的输入单词的信息。就图 8-2 的例子来说，输入“猫”时的 LSTM 层的输出（隐藏状态）受此时输入的单词“猫”的影响最大。因此，可以认为这个隐藏状态向量蕴含许多“猫的成分”。按照这样的理解，如图 8-3 所示，编码器输出的 $\boldsymbol{hs}$ 矩阵就可以视为各个单词对应的向量集合。

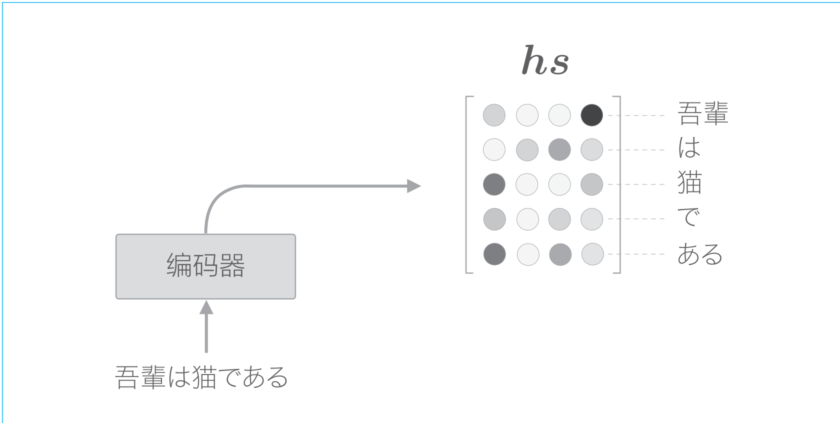


图8-3 编码器的输出 $\boldsymbol{hs}$ 具有和单词数相同数量的向量，各个向量中蕴含了各个单词对应的信息



因为编码器是从左向右处理的，所以严格来说，刚才的“猫”向量中含有“吾輩”“は”“猫”这3个单词的信息。考虑整体的平衡性，最好均衡地含有单词“猫”周围的信息。在这种情况下，从两个方向处理时序数据的双向RNN(或者双向LSTM)比较有效。我们后面再介绍双向RNN，这里先继续使用单向LSTM。

以上就是对编码器的改进。这里我们所做的改进只是将编码器的全部时刻的隐藏状态取出来而已。通过这个小改动，编码器可以根据输入语句的长度，成比例地编码信息。那么，解码器又将如何处理这个编码器的输出呢？接下来，我们对解码器进行改进。因为解码器的改进有许多值得讨论的地方，所以我们分3部分进行。

8.1.3 解码器的改进①

编码器整体输出各个单词对应的LSTM层的隐藏状态向量 $\boldsymbol{hs}$。然后，这个 $\boldsymbol{hs}$ 被传递给解码器，以进行时间序列的转换（图8-4）。

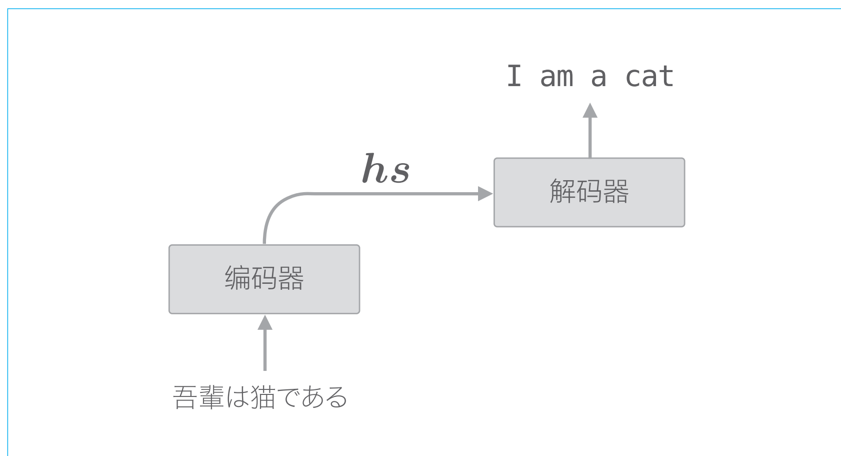


图8-4 编码器和解码器的关系

顺便说一下，在上一章的最简单的seq2seq中，仅将编码器最后的隐藏状态向量传递给了解码器。严格来说，这是将编码器的LSTM层的“最后”的隐藏状态放入了解码器的LSTM层的“最初”的隐藏状态。用图来表示的话，解码器的层结构如图8-5所示。

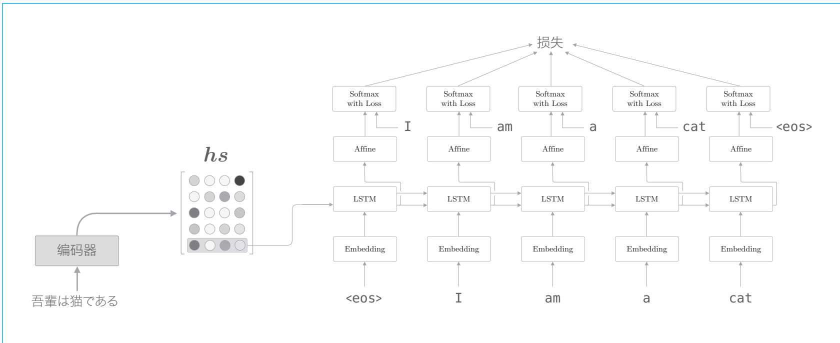


图 8-5　上一章的解码器的层结构（学习时）

如图 8-5 所示，上一章的解码器只用了编码器的 LSTM 层的最后的隐藏状态。如果使用 $\boldsymbol{hs}$，则只提取最后一行，再将其传递给解码器。下面我们改进解码器，以便能够使用全部 $\boldsymbol{hs}$。

我们在进行翻译时，大脑做了什么呢？比如，在将“吾輩は猫である”这句话翻译为英文时，肯定要用到诸如“吾輩 = I”“猫 = cat”这样的知识。也就是说，可以认为我们是专注于某个单词（或者单词集合），随时对这个单词进行转换的。那么，我们可以在 seq2seq 中重现同样的事情吗？确切地说，我们可以让 seq2seq 学习“输入和输出中哪些单词与哪些单词有关”这样的对应关系吗？



在机器翻译的历史中，很多研究都利用“猫 =cat”这样的单词对应关系的知识。这样的表示单词（或者词组）对应关系的信息称为对齐（alignment）。到目前为止，对齐主要是手工完成的，而我们将要介绍的 Attention 技术则成功地将对齐思想自动引入到了 seq2seq 中。这也是从“手工操作”到“机械自动化”的演变。

从现在开始，我们的目标是找出与“翻译目标词”有对应关系的“翻译源词”的信息，然后利用这个信息进行翻译。也就是说，我们的目标是仅关注必要的信息，并根据该信息进行时序转换。这个机制称为 Attention，是本章的主题。

在介绍 Attention 的细节之前，这里我们先给出它的整体框架。我们要实现的网络的层结构如图 8-6 所示。

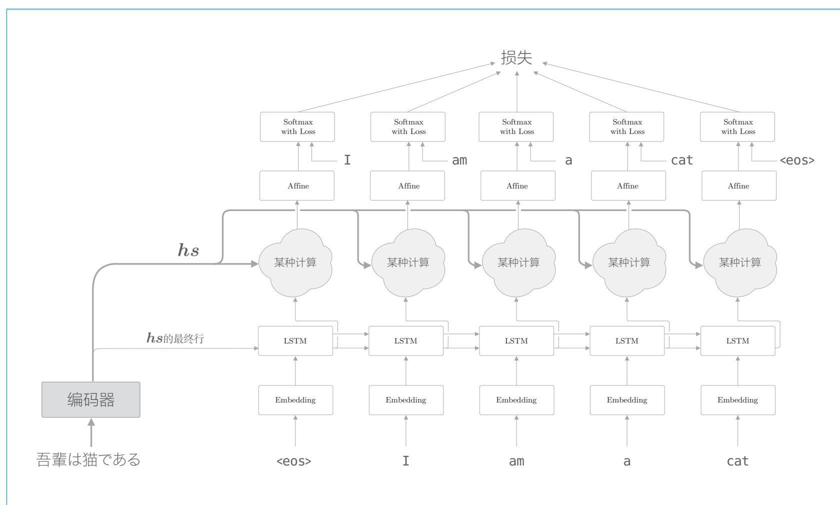


图 8-6　改进后的解码器的层结构

如图 8-6 所示，我们新增一个进行“某种计算”的层。这个“某种计算”接收（解码器）各个时刻的 LSTM 层的隐藏状态和编码器的 ***hs***。然后，从中选出必要的信息，并输出到 Affine 层。与之前一样，编码器的最后的隐藏状态向量传递给解码器最初的 LSTM 层。

图 8-6 的网络所做的工作是提取单词对齐信息。具体来说，就是从 ***hs*** 中选出与各个时刻解码器输出的单词有对应关系的单词向量。比如，当图 8-6 的解码器输出“I”时，从 ***hs*** 中选出“吾輩”的对应向量。也就是说，我们希望通过“某种计算”来实现这种选择操作。不过这里有个问题，就是选择（从多个事物中选取若干个）这一操作是无法进行微分的。



神经网络的学习一般通过误差反向传播法进行。因此，如果使用可微分的运算构造网络，就可以在误差反向传播法的框架内进行学习；而如果不使用可微分的运算，基本上也就没有办法使用误差反向传播法。

可否将“选择”这一操作换成可微分的运算呢？实际上，解决这个问题的思路很简单（但是，就像哥伦布蛋一样，第一个想到是很难的）。这个思路就是，与其“单选”，不如“全选”。如图 8-7 所示，我们另行计算表示各个单词重要度（贡献值）的权重。

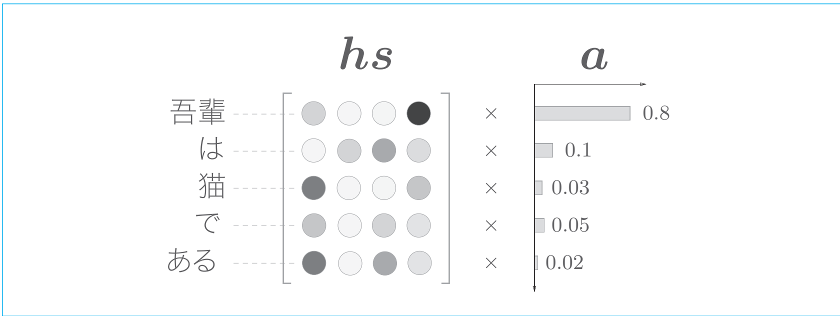


图8-7 对各个单词计算表示重要度的权重（后文介绍如何计算）

如图 8-7 所示，这里使用了表示各个单词重要度的权重（记为 $\boldsymbol{a}$）。此时，$\boldsymbol{a}$ 像概率分布一样，各元素是 0.0 ~ 1.0 的标量，总和是 1。然后，计算这个表示各个单词重要度的权重和单词向量 $\boldsymbol{hs}$ 的加权和，可以获得目标向量。这一系列计算如图 8-8 所示。

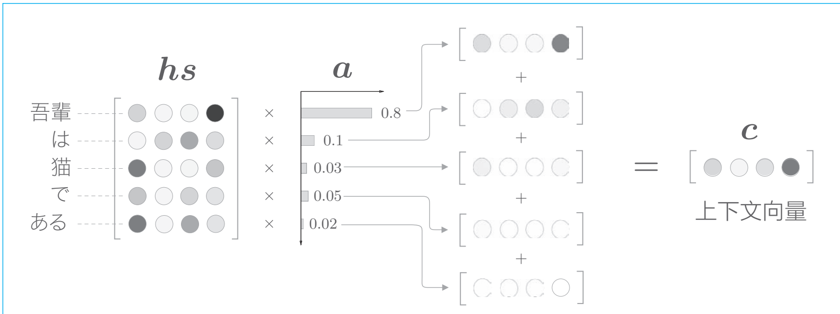


图8-8 通过计算加权和，可以得到上下文向量

如图 8-8 所示，计算单词向量的加权和，这里将结果称为上下文向量，并用符号 $\boldsymbol{c}$ 表示。顺便说一下，如果我们仔细观察，就可以发现“吾輩”对

应的权重为 0.8。这意味着上下文向量 $\boldsymbol{c}$ 中含有很多“吾輩”向量的成分，可以说这个加权和计算基本代替了“选择”向量的操作。假设“吾輩”对应的权重是 1，其他单词对应的权重是 0，那么这就相当于“选择”了“吾輩”向量。



上下文向量 $\boldsymbol{c}$ 中包含了当前时刻进行变换（翻译）所需的信息。更确切地说，模型要从数据中学习出这种能力。

下面，我们从代码的角度来看一下目前为止的内容。这里随意地生成编码器的输出 $\boldsymbol{hs}$ 和各个单词的权重 $\boldsymbol{a}$，并给出求它们的加权和的实现，代码如下所示，请注意多维数组的形状。

```
import numpy as np

T, H = 5, 4
hs = np.random.randn(T, H)
a = np.array([0.8, 0.1, 0.03, 0.05, 0.02])

ar = a.reshape(5, 1).repeat(4, axis=1)
print(ar.shape)
# (5, 4)

t = hs * ar
print(t.shape)
# (5, 4)

c = np.sum(t, axis=0)
print(c.shape)
# (4,)
```

设时序数据的长度 `T=5`，隐藏状态向量的元素个数 `H=4`，这里给出了加权和的计算过程。我们先关注代码 `ar = a.reshape(5, 1).repeat(4, axis=1)`。如图 8-9 所示，这行代码将 `a` 转化为 `ar`。

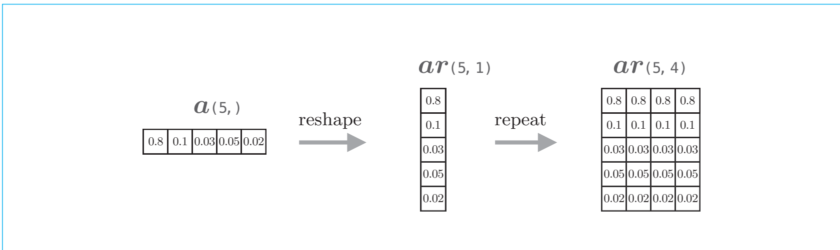


图8-9 通过reshape()和repeat()方法从a生成ar（变量的形状显示在其右侧）

如图 8-9 所示，我们要做的是复制形状为 `(5,)` 的 a，创建 `(5,4)` 的数组。因此，通过 `a.reshape(5, 1)` 将 a 的形状从 `(5,)` 转化为 `(5,1)`。然后，在第 1 个轴方向上（`axis=0`）重复这个变形后的数组 4 次，生成形状为 `(5,4)` 的数组。



`repeat()` 方法复制多维数组的元素生成新的多维数组。设 x 为 NumPy 多维数组，则可以像 `x.repeat(rep, axis)` 这样使用。这里参数 `rep` 指定复制的次数，`axis` 指定要进行复制的轴（维度）。比如，在 x 的形状为 `(X, Y, Z)` 的情况下，`x.repeat(3, axis=1)` 沿 x 的第 1 个轴方向（第 1 个维度）进行复制，生成形状为 `(X, 3*Y, Z)` 的多维数组。

此外，这里也可以不使用 `repeat()` 方法，而使用 NumPy 的广播功能。此时，令 `ar = a.reshape(5, 1)`，然后计算 `hs * ar`。如图 8-10 所示，ar 会自动扩展以匹配 hs 的形状。

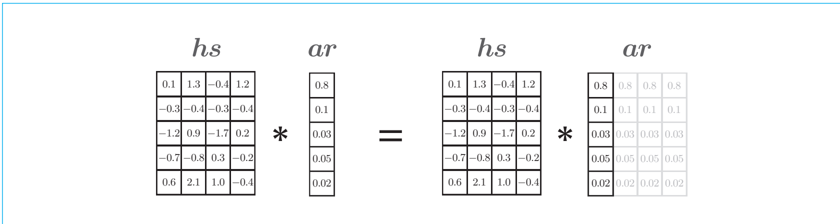


图8-10 NumPy的广播

为了提高执行效率，这里应该使用 NumPy 的广播，而不是 `repeat()` 方法。但是，在这种情况下，需要注意的是，在许多我们看不见的地方多维数

组的元素被复制了。由 1.3.4.3 节可知，这相当于计算图中的 Repeat 节点。因此，在反向传播时，需要执行 Repeat 节点的反向传播。

如图 8-10 所示，先计算对应元素的乘积，然后通过 `c = np.sum(hs*ar, axis=0)` 求和。这里，通过参数 `axis` 可以指定在哪个轴方向（维度）上求和。如果我们注意一下数组的形状，`axis` 的使用方法就会很清楚。比如，当 `x` 的形状为 `(X, Y, Z)` 时，`np.sum(x, axis=1)` 的输出（和）的形状为 `(X, Z)`。这里的重点是，求和会使一个轴“消失”。在上面的例子中，`hs*ar` 的形状为 `(5,4)`，通过消除第 0 个轴，获得了形状为 `(4,)` 的矩阵（向量）。



计算加权和最简单有效的方法是使用矩阵乘积。就上面的例子来说，只需要 `np.dot(a, hs)` 这一行代码就可以获得目标结果。不过，这样只能处理一笔数据（样本），很难将其扩展到批处理。如果非要扩展，就需要用到“张量积”，这会使事情变得有些复杂（在这种情况下，需要使用 `np.tensordot()` 和 `np.einsum()` 方法）。简单起见，这里我们不使用矩阵乘积，而是通过 `repeat()` 和 `sum()` 方法来实现加权和的计算。

下面进行批处理版的加权和的实现，具体如下所示（这里随机创建 `hs` 和 `a`）。

```
N, T, H = 10, 5, 4
hs = np.random.randn(N, T, H)
a = np.random.randn(N, T)
ar = a.reshape(N, T, 1).repeat(H, axis=2)
# ar = a.reshape(N, T, 1) # 使用广播

t = hs * ar
print(t.shape)
# (10, 5, 4)

c = np.sum(t, axis=1)
print(c.shape)
# (10, 4)
```

这里的批处理与之前的实现几乎一样。只要注意数组的形状，应该很快就能确定 `repeat()` 和 `sum()` 需要指定的维度（轴）。作为总结，我们把加权和的计算用计算图表示出来（图 8-11）。

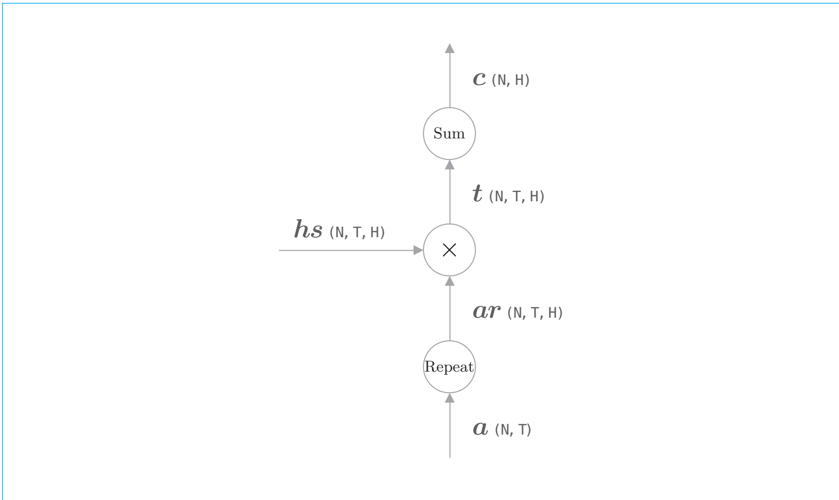


图 8-11　加权和的计算图

如图 8-11 所示，这里使用 Repeat 节点复制 a。之后，通过“×”节点计算对应元素的乘积，通过 Sum 节点求和。现在考虑这个计算图的反向传播。其实，所需要的知识都已经齐备。第 1 章介绍了 Repeat 节点和 Sum 节点的反向传播。这里重述一下要点：“Repeat 的反向传播是 Sum”“Sum 的反向传播是 Repeat”。只要注意到张量的形状，就不难知道应该对哪个轴进行 Sum，对哪个轴进行 Repeat。

现在我们将图 8-11 的计算图实现为层，这里称之为 Weight Sum 层，其实现如下所示（✏ ch08/attention_layer.py）。

```
class WeightSum:
    def __init__(self):
        self.params, self.grads = [], []
        self.cache = None
```

```
    def forward(self, hs, a):
        N, T, H = hs.shape

        ar = a.reshape(N, T, 1).repeat(H, axis=2)
        t = hs * ar
        c = np.sum(t, axis=1)

        self.cache = (hs, ar)
        return c

    def backward(self, dc):
        hs, ar = self.cache
        N, T, H = hs.shape

        dt = dc.reshape(N, 1, H).repeat(T, axis=1)  # sum的反向传播
        dar = dt * hs
        dhs = dt * ar
        da = np.sum(dar, axis=2)  # repeat的反向传播

        return dhs, da
```

以上就是计算上下文向量的 Weight Sum 层的实现。因为这个层没有要学习的参数，所以根据本书的代码规范，此处为 `self.params = []`。其他应该没有特别难的地方，我们继续往下看。

8.1.4 解码器的改进②

有了表示各个单词重要度的权重 $\boldsymbol{a}$，就可以通过加权和获得上下文向量。那么，怎么求这个 $\boldsymbol{a}$ 呢？当然不需要我们手动指定，我们只需要做好让模型从数据中自动学习它的准备工作。

下面我们来看一下各个单词的权重 $\boldsymbol{a}$ 的求解方法。首先，从编码器的处理开始到解码器第一个 LSTM 层输出隐藏状态向量的处理为止的流程如图 8-12 所示。

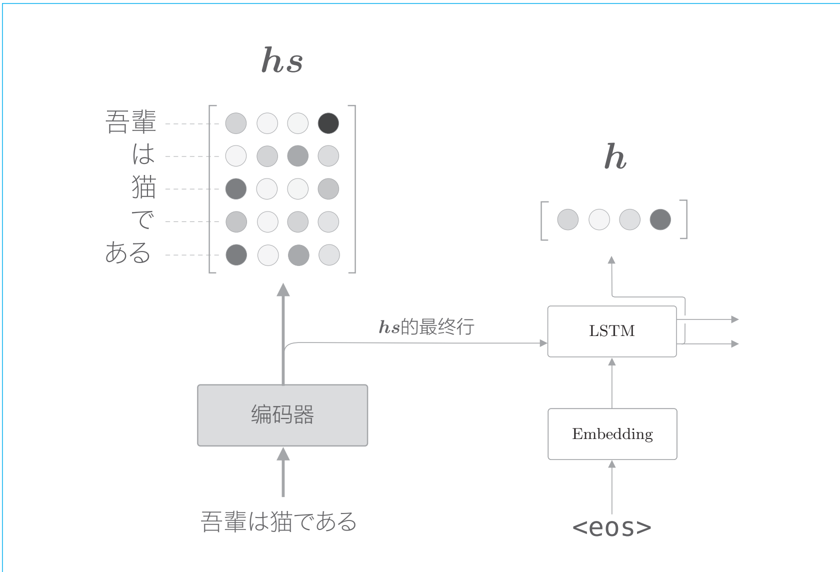


图 8-12　解码器第 1 个 LSTM 层的隐藏状态向量

在图 8-12 中，用 $\boldsymbol{h}$ 表示解码器的 LSTM 层的隐藏状态向量。此时，我们的目标是用数值表示这个 $\boldsymbol{h}$ 在多大程度上和 $\boldsymbol{hs}$ 的各个单词向量“相似”。有几种方法可以做到这一点，这里我们使用最简单的向量内积。顺便说一下，向量 $\boldsymbol{a} = (a_1, a_2, \cdots, a_n)$ 和向量 $\boldsymbol{b} = (b_1, b_2, \cdots, b_n)$ 的内积为：

$$\boldsymbol{a} \cdot \boldsymbol{b} = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n \tag{8.1}$$

式 (8.1) 的含义是两个向量在多大程度上指向同一方向，因此使用内积作为两个向量的“相似度”是非常自然的选择。



计算向量相似度的方法有好几种。除了内积之外，还有使用小型的神经网络输出得分的做法。文献 [49] 中提出了几种输出得分的方法。

下面用图表示基于内积计算向量间相似度的处理流程（图 8-13）。

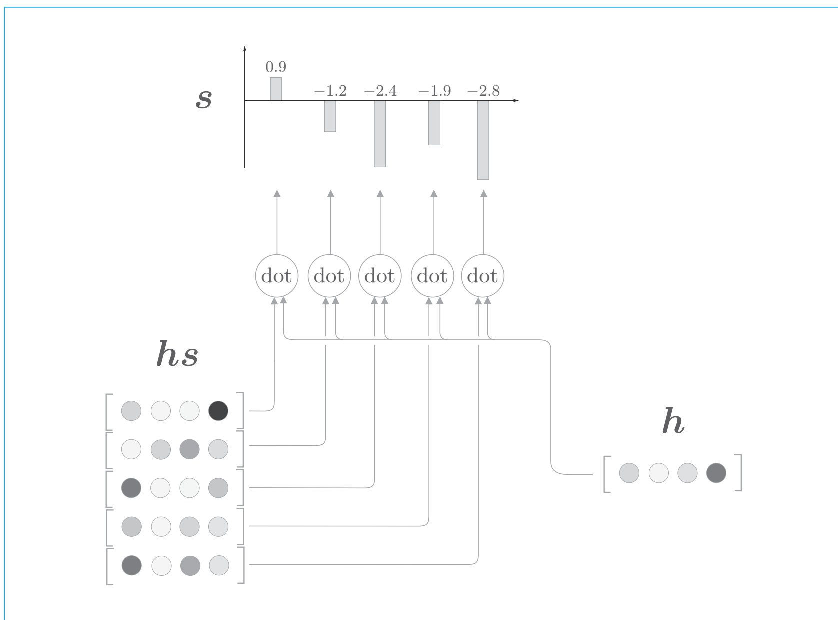


图 8-13 基于内积计算 $\boldsymbol{hs}$ 的各行与 $\boldsymbol{h}$ 的相似度（内积用 dot 节点表示）

如图 8-13 所示，这里通过向量内积算出 $\boldsymbol{h}$ 和 $\boldsymbol{hs}$ 的各个单词向量之间的相似度，并将其结果表示为 $\boldsymbol{s}$。不过，这个 $\boldsymbol{s}$ 是正规化之前的值，也称为得分。接下来，使用老一套的 Softmax 函数对 $\boldsymbol{s}$ 进行正规化（图 8-14）。

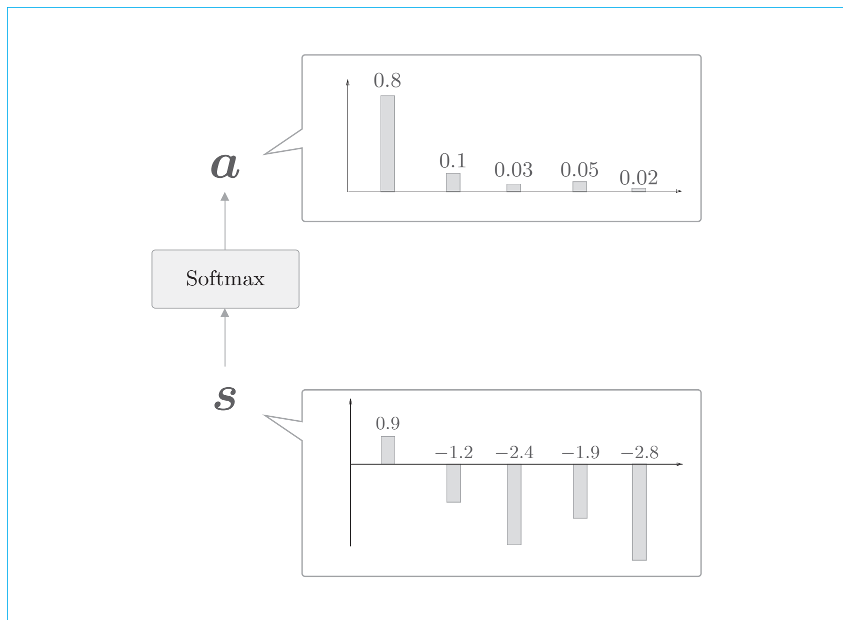


图8-14　基于Softmax的正规化

使用 Softmax 函数之后，输出的 $\boldsymbol{a}$ 的各个元素的值在 0.0 ~ 1.0，总和为 1，这样就求得了表示各个单词权重的 $\boldsymbol{a}$。现在我们从代码角度来看一下这些处理。

```
import sys
sys.path.append('..')
from common.layers import Softmax
import numpy as np

N, T, H = 10, 5, 4
hs = np.random.randn(N, T, H)
h = np.random.randn(N, H)
hr = h.reshape(N, 1, H).repeat(T, axis=1)
# hr = h.reshape(N, 1, H)  # 广播

t = hs * hr
print(t.shape)
# (10, 5, 4)
```

```
s = np.sum(t, axis=2)
print(s.shape)
# (10, 5)

softmax = Softmax()
a = softmax.forward(s)
print(a.shape)
# (10, 5)
```

以上就是进行批处理的代码。如前所述，此处我们通过 reshape() 和 repeat() 方法生成形状合适的 hr。在使用 NumPy 的广播的情况下，不需要 repeat()。此时的计算图如图 8-15 所示。

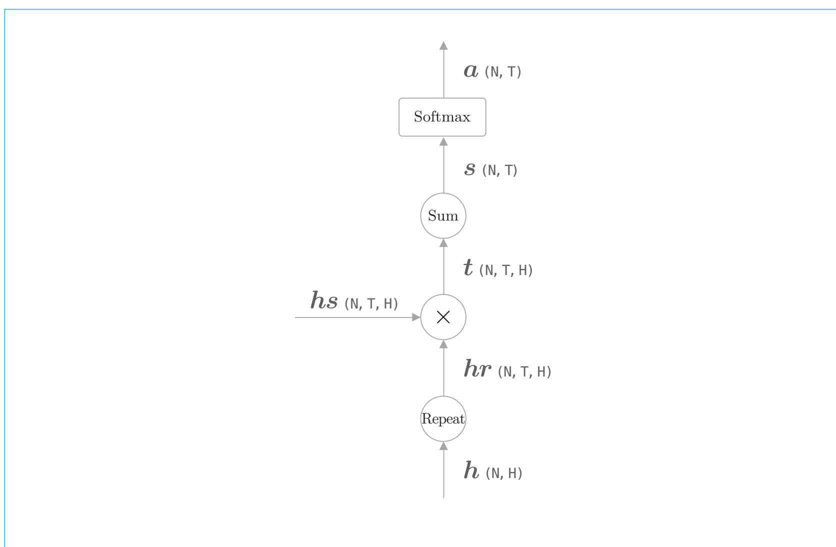


图 8-15　计算各个单词权重的计算图

如图 8-15 所示，这里的计算图由 Repeat 节点、表示对应元素的乘积的“×”节点、Sum 节点和 Softmax 层构成。我们将这个计算图表示的处理实现为 AttentionWeight 类（✏ ch08/attention_layer.py）。

```
import sys
sys.path.append('..')
```

```
from common.np import * # import numpy as np
from common.layers import Softmax

class AttentionWeight:
    def __init__(self):
        self.params, self.grads = [], []
        self.softmax = Softmax()
        self.cache = None

    def forward(self, hs, h):
        N, T, H = hs.shape

        hr = h.reshape(N, 1, H).repeat(T, axis=1)
        t = hs * hr
        s = np.sum(t, axis=2)
        a = self.softmax.forward(s)

        self.cache = (hs, hr)
        return a

    def backward(self, da):
        hs, hr = self.cache
        N, T, H = hs.shape

        ds = self.softmax.backward(da)
        dt = ds.reshape(N, T, 1).repeat(H, axis=2)
        dhs = dt * hr
        dhr = dt * hs
        dh = np.sum(dhr, axis=1)

        return dhs, dh
```

类似于之前的 Weight Sum 层，这个实现有 Repeat 和 Sum 运算。只要注意到这两个运算的反向传播，其他应该就没有特别难的地方。下面，我们进行解码器的最后一个改进。

8.1.5 解码器的改进③

在此之前，我们分两节介绍了解码器的改进方案。8.1.3 节和 8.1.4 节分别实现了 Weight Sum 层和 Attention Weight 层。现在，我们将这两层组合起来，结果如图 8-16 所示。

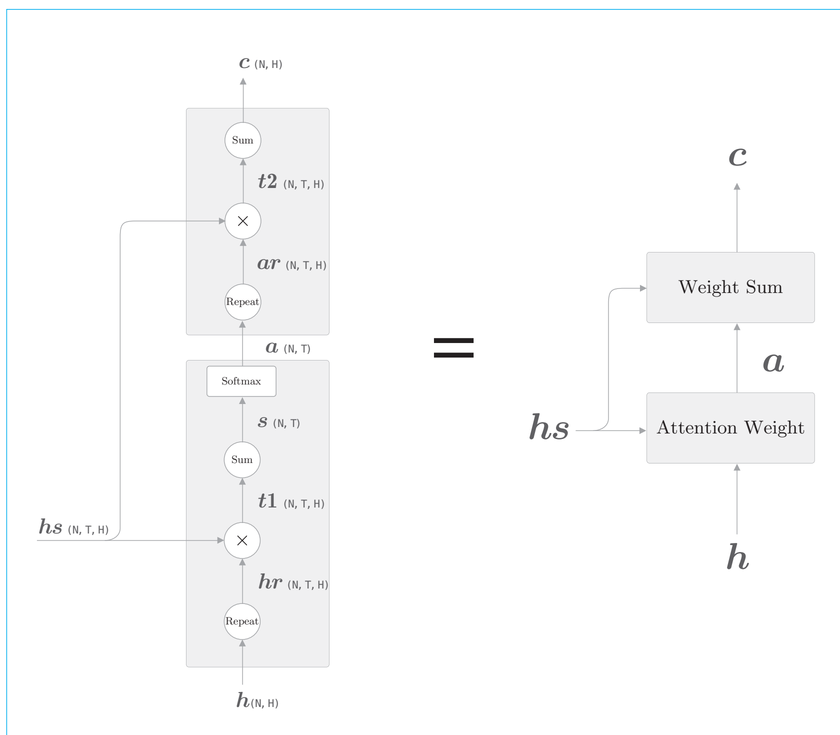


图 8-16 计算上下文向量的计算图

图 8-16 显示了用于获取上下文向量 $\boldsymbol{c}$ 的计算图的全貌。我们已经分为 Weight Sum 层和 Attention Weight 层进行了实现。重申一下，这里进行的计算是：Attention Weight 层关注编码器输出的各个单词向量 $\boldsymbol{hs}$，并计算各个单词的权重 $\boldsymbol{a}$；然后，Weight Sum 层计算 $\boldsymbol{a}$ 和 $\boldsymbol{hs}$ 的加权和，并输出上下文向量 $\boldsymbol{c}$。我们将进行这一系列计算的层称为 Attention 层（图 8-17）。

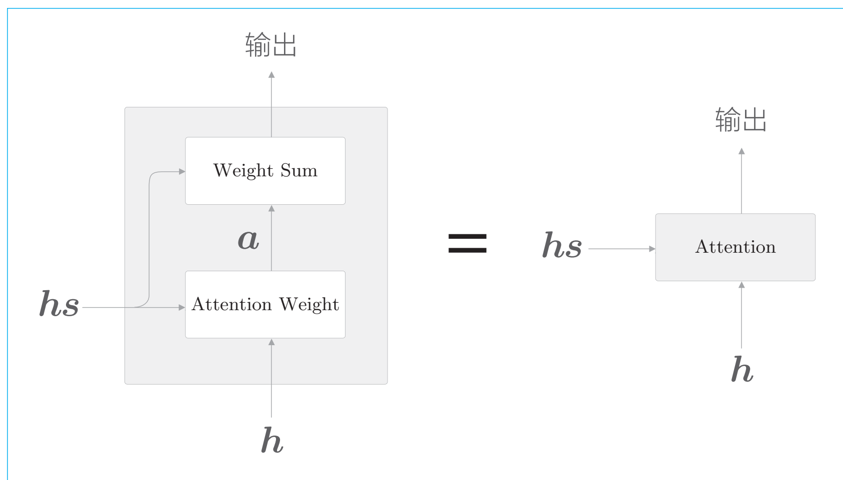


图8-17 将左边的计算图整合为Attention层

以上就是 Attention 技术的核心内容。关注编码器传递的信息 $\boldsymbol{hs}$ 中的重要元素，基于它算出上下文向量，再传递给上一层（这里，Affine 层在上一层等待）。下面给出 Attention 层的实现（✏ ch08/attention_layer.py）。

```
class Attention:
    def __init__(self):
        self.params, self.grads = [], []
        self.attention_weight_layer = AttentionWeight()
        self.weight_sum_layer = WeightSum()
        self.attention_weight = None

    def forward(self, hs, h):
        a = self.attention_weight_layer.forward(hs, h)
        out = self.weight_sum_layer.forward(hs, a)
        self.attention_weight = a
        return out

    def backward(self, dout):
        dhs0, da = self.weight_sum_layer.backward(dout)
        dhs1, dh = self.attention_weight_layer.backward(da)
        dhs = dhs0 + dhs1
        return dhs, dh
```

以上是 Weight Sum 层和 Attention Weight 层的正向传播和反向传播。为了以后可以访问各个单词的权重，这里设定成员变量 `attention_weight`，如此就完成了 Attention 层的实现。我们将这个 Attention 层放在 LSTM 层和 Affine 层的中间，如图 8-18 所示。

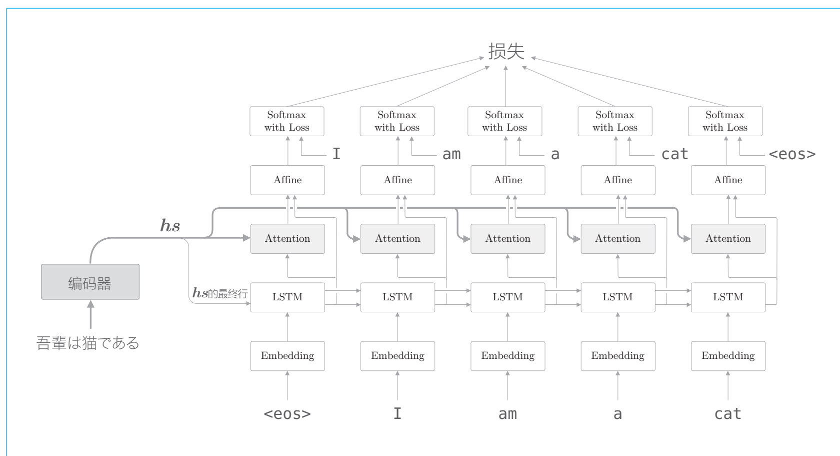


图 8-18　具有 Attention 层的解码器的层结构

如图 8-18 所示，编码器的输出 ***hs*** 被输入到各个时刻的 Attention 层。另外，这里将 LSTM 层的隐藏状态向量输入 Affine 层。根据上一章的解码器的改进，可以说这个扩展非常自然。如图 8-19 所示，我们将 Attention 信息“添加”到了上一章的解码器上。

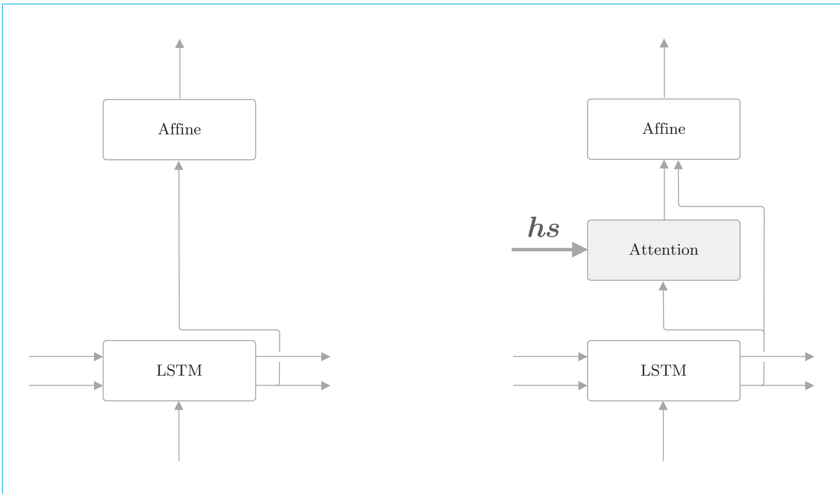


图8-19 上一章的解码器（左图）和带Attention的解码器（右图）的比较：选出从LSTM层到Affine层的部分

如图 8-19 所示，我们向上一章的解码器“添加”基于 Attention 层的上下文向量信息。因此，除了将原先的 LSTM 层的隐藏状态向量传给 Affine 层之外，追加输入 Attention 层的上下文向量。



在图 8-19 中，上下文向量和隐藏状态向量这两个向量被输入 Affine 层。如前所述，这意味着将这两个向量拼接起来，将拼接后的向量输入 Affine 层。

最后，我们将在图 8-18 的时序方向上扩展的多个 Attention 层整体实现为 Time Attention 层，如图 8-20 所示。

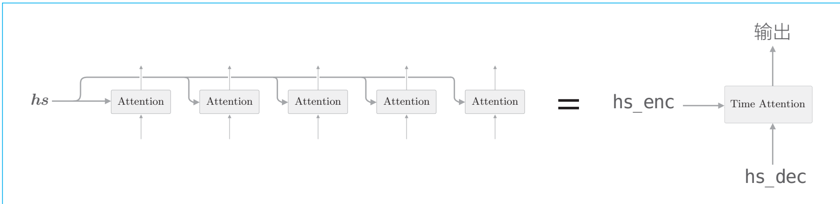


图8-20 将多个Attention层整体实现为Time Attention层

由图 8-20 可知，Time Attention 层只是组合了多个 Attention 层，其实现如下所示（✏ ch08/attention_layer.py）。

```
class TimeAttention:
    def __init__(self):
        self.params, self.grads = [], []
        self.layers = None
        self.attention_weights = None

    def forward(self, hs_enc, hs_dec):
        N, T, H = hs_dec.shape
        out = np.empty_like(hs_dec)
        self.layers = []
        self.attention_weights = []

        for t in range(T):
            layer = Attention()
            out[:, t, :] = layer.forward(hs_enc, hs_dec[:,t,:])
            self.layers.append(layer)
            self.attention_weights.append(layer.attention_weight)

        return out

    def backward(self, dout):
        N, T, H = dout.shape
        dhs_enc = 0
        dhs_dec = np.empty_like(dout)

        for t in range(T):
            layer = self.layers[t]
            dhs, dh = layer.backward(dout[:, t, :])
            dhs_enc += dhs
            dhs_dec[:,t,:] = dh

        return dhs_enc, dhs_dec
```

这里仅创建必要数量的 Attention 层（代码中为 T 个），各自进行正向传播和反向传播。另外，attention_weights 列表中保存了各个 Attention 层对各个单词的权重。

以上，我们介绍了Attention的结构及其实现。下面我们使用Attention来实现 seq2seq，并尝试挑战一个真实问题，以确认 Attention 的效果。

8.2 带Attention的seq2seq的实现

上一节实现了 Attention 层（以及 Time Attention 层），现在我们使用这个层来实现“带Attention的seq2seq”。和上一章实现了3个类（Encoder、Decoder 和 seq2seq）一样，这里我们也分别实现 3 个类（AttentionEncoder、AttentionDecoder 和 AttentionSeq2seq）。

8.2.1 编码器的实现

首先实现 AttentionEncoder 类。这个类和上一章实现的 Encoder 类几乎一样，唯一的区别是，Encoder 类的 forward() 方法仅返回 LSTM 层的最后的隐藏状态向量，而AttentionEncoder类则返回所有的隐藏状态向量。因此，这里我们继承上一章的 Encoder 类进行实现。AttentionEncoder 类的实现如下所示（✏ ch08/attention_seq2seq.py）。

```
import sys
sys.path.append('..')
from common.time_layers import *
from ch07.seq2seq import Encoder, Seq2seq
from ch08.attention_layer import TimeAttention


class AttentionEncoder(Encoder):
    def forward(self, xs):
        xs = self.embed.forward(xs)
        hs = self.lstm.forward(xs)
        return hs

    def backward(self, dhs):
        dout = self.lstm.backward(dhs)
        dout = self.embed.backward(dout)
        return dout
```

8.2.2 解码器的实现

接着实现使用了 Attention 层的解码器。使用了 Attention 的解码器的层结构如图 8-21 所示。

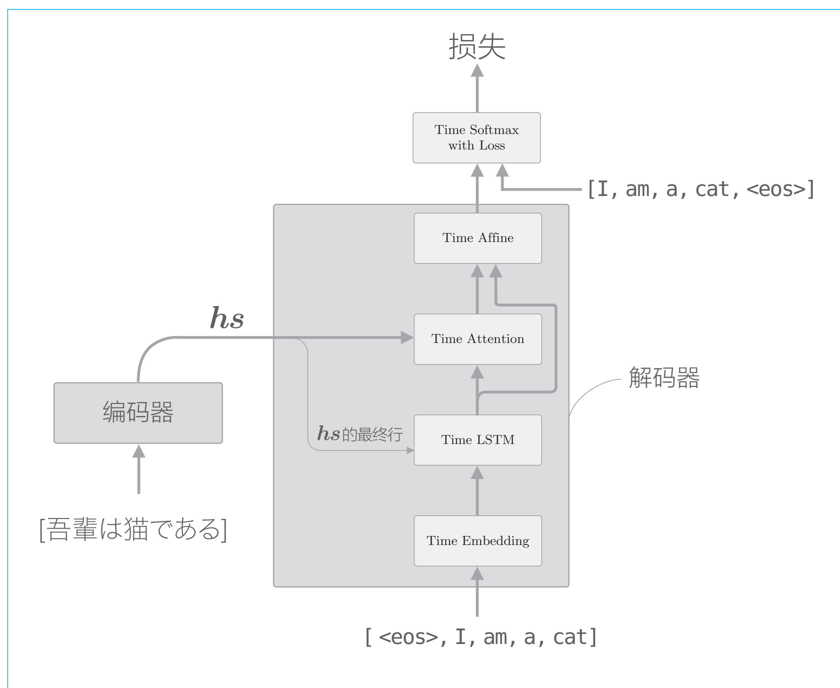


图 8-21 解码器的层结构

从图 8-21 中可以看出，和上一章的实现一样，Softmax 层（更确切地说，是 Time Softmax with Loss 层）之前的层都作为解码器。另外，和上一章一样，除了正向传播 `forward()` 方法和反向出传播 `backward()` 方法之外，还实现了生成新单词序列（字符序列）的 `generate()` 方法。这里仅给出 Attention Decoder 层的初始化方法和 `forward()` 方法的实现，如下所示（✏ `ch08/attention_seq2seq.py`）。

```
class AttentionDecoder:
    def __init__(self, vocab_size, wordvec_size, hidden_size):
        V, D, H = vocab_size, wordvec_size, hidden_size
        rn = np.random.randn

        embed_W = (rn(V, D) / 100).astype('f')
        lstm_Wx = (rn(D, 4 * H) / np.sqrt(D)).astype('f')
        lstm_Wh = (rn(H, 4 * H) / np.sqrt(H)).astype('f')
        lstm_b = np.zeros(4 * H).astype('f')
        affine_W = (rn(2*H, V) / np.sqrt(2*H)).astype('f')
        affine_b = np.zeros(V).astype('f')

        self.embed = TimeEmbedding(embed_W)
        self.lstm = TimeLSTM(lstm_Wx, lstm_Wh, lstm_b, stateful=True)
        self.attention = TimeAttention()
        self.affine = TimeAffine(affine_W, affine_b)
        layers = [self.embed, self.lstm, self.attention, self.affine]

        self.params, self.grads = [], []
        for layer in layers:
            self.params += layer.params
            self.grads += layer.grads

    def forward(self, xs, enc_hs):
        h = enc_hs[:,-1]
        self.lstm.set_state(h)

        out = self.embed.forward(xs)
        dec_hs = self.lstm.forward(out)
        c = self.attention.forward(enc_hs, dec_hs)
        out = np.concatenate((c, dec_hs), axis=2)
        score = self.affine.forward(out)

        return score

    def backward(self, dscore):
        # 参照源代码

    def generate(self, enc_hs, start_id, sample_size):
        # 参照源代码
```

这里的实现除使用了新的 Time Attention 层之外，和上一章的 `Decoder` 类没有什么太大的不同。需要注意的是，`forward()` 方法中拼接了 Time Attention 层的输出和 LSTM 层的输出。在上面的代码中，使用 `np.concatenate()` 方法进行拼接。

这里省略对 `AttentionDecoder` 类的 `backward()` 和 `generate()` 方法的说明。最后，我们使用 `AttentionEncoder` 类和 `AttentionDecoder` 类来实现 `AttentionSeq2seq` 类。

8.2.3 seq2seq 的实现

`AttentionSeq2seq` 类的实现也和上一章实现的 seq2seq 几乎一样。区别仅在于，编码器使用 `AttentionEncoder` 类，解码器使用 `AttentionDecoder` 类。因此，只要继承上一章的 `Seq2seq` 类，并改一下初始化方法，就可以实现 `AttentionSeq2seq` 类（✏ ch08/attention_seq2seq.py）。

```
from ch07.seq2seq import Encoder, Seq2seq

class AttentionSeq2seq(Seq2seq):
    def __init__(self, vocab_size, wordvec_size, hidden_size):
        args = vocab_size, wordvec_size, hidden_size
        self.encoder = AttentionEncoder(*args)
        self.decoder = AttentionDecoder(*args)
        self.softmax = TimeSoftmaxWithLoss()

        self.params = self.encoder.params + self.decoder.params
        self.grads = self.encoder.grads + self.decoder.grads
```

以上就是带 Attention 的 seq2seq 的实现。

8.3 Attention 的评价

下面，我们使用上一节实现的 `AttentionSeq2seq` 类来挑战一个实际问题。原本我们应该通过研究翻译问题来确认 Attention 的效果，可惜没能找到大小适中的翻译数据集。因此，我们转而通过研究“日期格式转换”问题（本

质上属于人为创造的问题，数据量有限），来确认带 Attention 的 seq2seq 的效果。



WMT 是一个有名的翻译数据集。这个数据集中提供了英语和法语（或者英语和德语）的平行学习数据。WMT 数据集在许多研究中都被作为基准使用，经常用于评价 seq2seq 的性能，不过它的数据量很大（超过 20 GB），使用起来不是很方便。

8.3.1 日期格式转换问题

这里我们要处理的是日期格式转换问题。这个任务旨在将使用英语的国家和地区所使用的各种各样的日期格式转换为标准格式。例如，将人写的“september 27, 1994”这样的日期数据转换为“1994-09-27”这样的标准格式，如图 8-22 所示。

september 27, 1994	→	1994-09-27
JUN 17, 2013	→	2013-06-17
2/10/93	→	1993-02-10

图 8-22 日期格式转换的例子

这里采用日期格式转换问题的原因有两个。首先，该问题并不像看上去那么简单。因为输入的日期数据存在各种各样的版本，所以转换规则也相应地复杂。如果尝试将这些转换规则全部写出来，那将非常费力。

其次，该问题的输入（问句）和输出（回答）存在明显的对应关系。具体而言，存在年月日的对应关系。因此，我们可以确认 Attention 有没有正确地关注各自的对应元素。

我们事先在 `dataset/date.txt` 中准备好了要处理的日期转换数据。如图 8-23 所示，这个文本文件包含 50 000 个日期转换用的学习数据。

```
 1 september 27, 1994          _1994-09-27
 2 August 19, 2003             _2003-08-19
 3 2/10/93                     _1993-02-10
 4 10/31/90                    _1990-10-31
 5 TUESDAY, SEPTEMBER 25, 1984 _1984-09-25
 6 JUN 17, 2013                _2013-06-17
 7 april 3, 1996               _1996-04-03
 8 October 24, 1974            _1974-10-24
 9 AUGUST 11, 1986             _1986-08-11
10 February 16, 2015           _2015-02-16
11 October 12, 1988            _1988-10-12
12 6/3/73                      _1973-06-03
13 Sep 30, 1981                _1981-09-30
14 June 19, 1977               _1977-06-19
15 OCTOBER 22, 2005            _2005-10-22
Lines: 50,000   Chars: 2,050,000                2.05 MB
```

图8-23　用于日期格式转换的学习数据：空格显示为灰点

为了对齐输入语句的长度，本书提供的日期数据集填充了空格，并将“_”（下划线）设置为输入和输出的分隔符。另外，因为这个问题输出的字符数是恒定的，所以无须使用分隔符来指示输出的结束。



如上一章所述，本书提供了一个可以轻松处理上述seq2seq用的学习数据的Python模块，这个模块位于`dataset/sequence.py`中。

8.3.2　带Attention的seq2seq的学习

下面，我们在日期转换用的数据集上进行`AttentionSeq2seq`的学习，学习用的代码如下所示（✏ `ch08/train.py`）。

```
import sys
sys.path.append('..')
import numpy as np
from dataset import sequence
from common.optimizer import Adam
from common.trainer import Trainer
from common.util import eval_seq2seq
```

```
from attention_seq2seq import AttentionSeq2seq
from ch07.seq2seq import Seq2seq
from ch07.peeky_seq2seq import PeekySeq2seq


# 读入数据
(x_train, t_train), (x_test, t_test) = sequence.load_data('date.txt')
char_to_id, id_to_char = sequence.get_vocab()

# 反转输入语句
x_train, x_test = x_train[:, ::-1], x_test[:, ::-1]

# 设定超参数
vocab_size = len(char_to_id)
wordvec_size = 16
hidden_size = 256
batch_size = 128
max_epoch = 10
max_grad = 5.0

model = AttentionSeq2seq(vocab_size, wordvec_size, hidden_size)
optimizer = Adam()
trainer = Trainer(model, optimizer)

acc_list = []
for epoch in range(max_epoch):
    trainer.fit(x_train, t_train, max_epoch=1,
                batch_size=batch_size, max_grad=max_grad)

    correct_num = 0
    for i in range(len(x_test)):
        question, correct = x_test[[i]], t_test[[i]]
        verbose = i < 10
        correct_num += eval_seq2seq(model, question, correct,
                                    id_to_char, verbose, is_reverse=True)

    acc = float(correct_num) / len(x_test)
    acc_list.append(acc)
    print('val acc %.3f%%' % (acc * 100))


model.save_params()
```

这里显示的代码和上一章的加法问题的学习用代码几乎一样。区别在于，它读入日期数据作为学习数据，使用`AttentionSeq2seq`作为模型。另外，这里还使用了反转输入语句的技巧（Reverse）。之后，在学习的同时，每个 epoch 使用测试数据计算正确率。为了查看结果，我们将前 10 个问题的问句和回答输出到终端。

现在我们运行一下上面的代码。随着学习的进行，结果如图 8-24 所示。

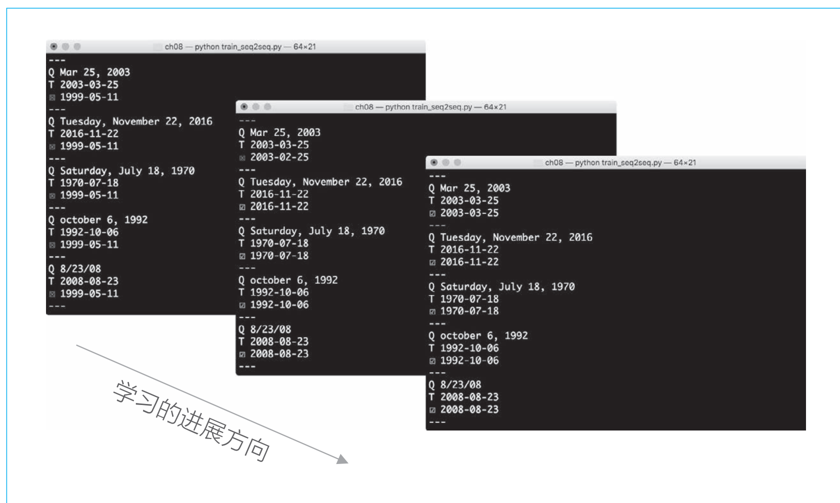


图 8-24　显示在终端上的结果的演变

如图 8-24 所示，随着学习的深入，带 Attention 的 seq2seq 变聪明了。实际上，没过多久，它就对大多数问题给出了正确答案。此时，测试数据的正确率（代码中的`acc_list`）如图 8-25 所示。

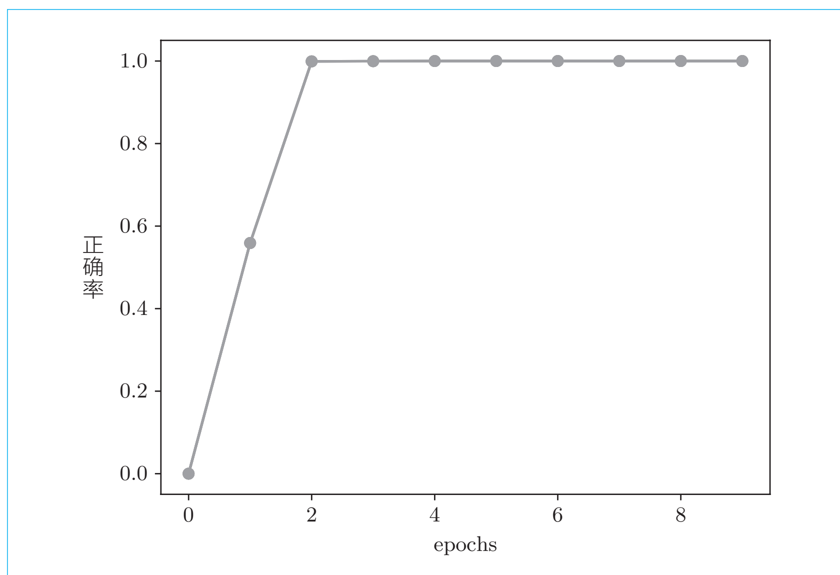


图 8-25 正确率的演变

如图 8-25 所示，从第 1 个 epoch 开始，正确率迅速上升，到第 2 个 epoch 时，几乎可以正确回答所有问题。这可以说是一个很好的结果。我们将这个结果与上一章的模型比较一下，如图 8-26 所示。

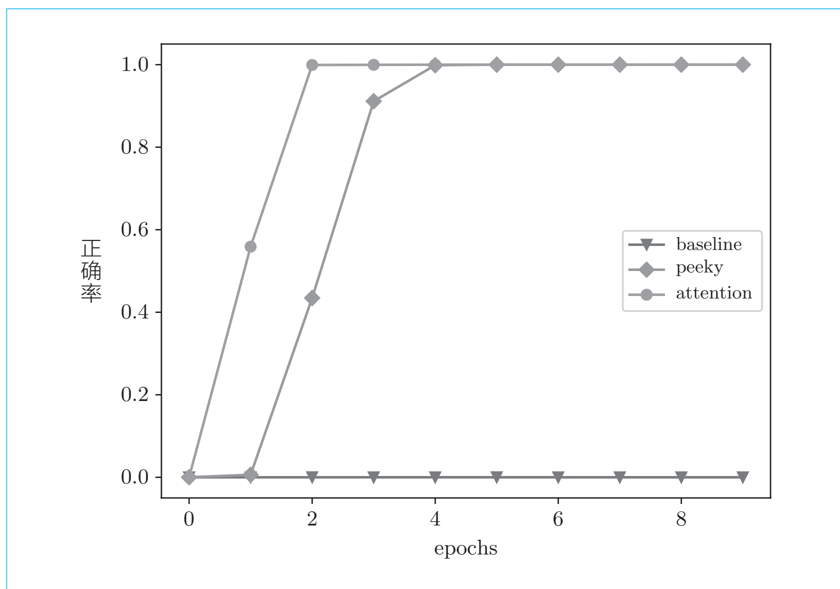


图8-26　与其他模型的比较：图中的baseline是上一章中的简单的seq2seq，peeky是使用“偷窥”技术改进过的seq2seq（所有的模型都反转了输入语句）

从图 8-26 的结果可知，简单的 seq2seq（图中的 baseline）完全没法用。即使经过了 10 个 epoch，大多数问题还是不能回答正确。而使用了“偷窥”技术的 Peeky 给出了良好的结果，从第 3 个 epoch 开始，模型的正确率开始上升，在第 4 个 epoch 时，正确率达到了 100%。但是，就学习速度而言，Attention 稍微有些优势。

在这次的实验中，就最终精度来看，Attention 和 Peeky 取得了差不多的结果。但是，随着时序数据变长、变复杂，除了学习速度之外，Attention 在精度上也会变得更有优势。

8.3.3　Attention的可视化

接下来，我们对 Attention 进行可视化。在进行时序转换时，实际观察 Attention 在注意哪个元素。因为在 Attention 层中，各个时刻的 Attention 权重均保存到了成员变量中，所以我们可以轻松地进行可视化。

在我们的实现中，Time Attention 层中的成员变量 `attention_weights` 保存了各个时刻的 Attention 权重，据此可以将输入语句和输出语句的各个单词的对应关系绘制成一张二维地图。这里，我们针对学习好的 `AttentionSeq2seq`，对进行日期格式转换时的 Attention 权重进行可视化。此处，我们不给出代码，仅将结果显示在图 8-27 上（✏ `ch08/visualize_attention.py`）。

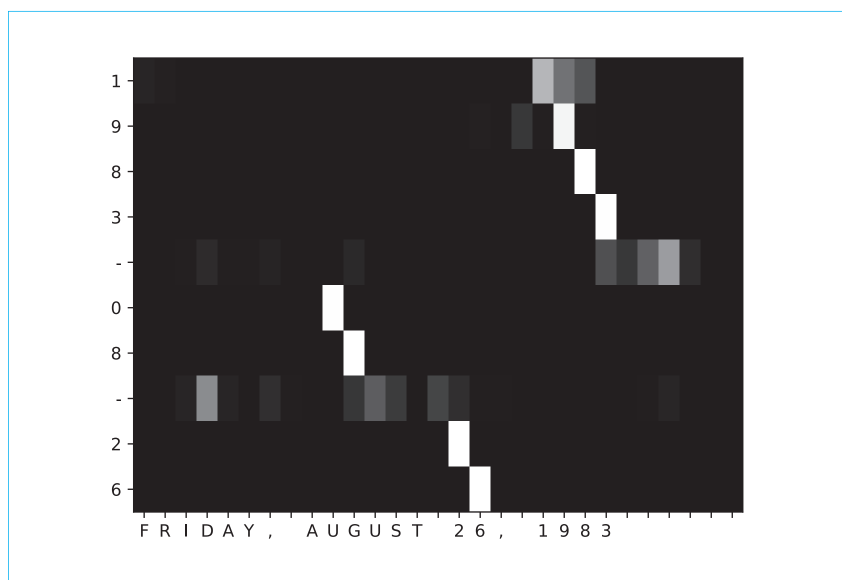


图8-27　使用学习好的模型，对进行时序转换时的 Attention 权重进行可视化。横轴是模型的输入语句，纵轴是模型的输出语句。地图中的元素越接近白色，其值越大（接近1）

图 8-27 是 seq2seq 进行时序转换时的 Attention 权重的可视化结果。例如，我们可以看到，当 seq2seq 输出第 1 个“1”时，注意力集中在输入语句的“1”上。这里需要特别注意年月日的对应关系。仔细观察图中的结果，纵轴（输出）的“1983”和“26”恰好对应于横轴（输入）的“1983”和“26”。另外，输入语句的“AUGUST”对应于表示月份的“08”，这一点也很令人惊讶。这表明 seq2seq 从数据中学习到了“August”和“8 月”的对应关系。图 8-28 中给出了其他一些例子，从中也可以很清楚地看到年月日的对应关系。

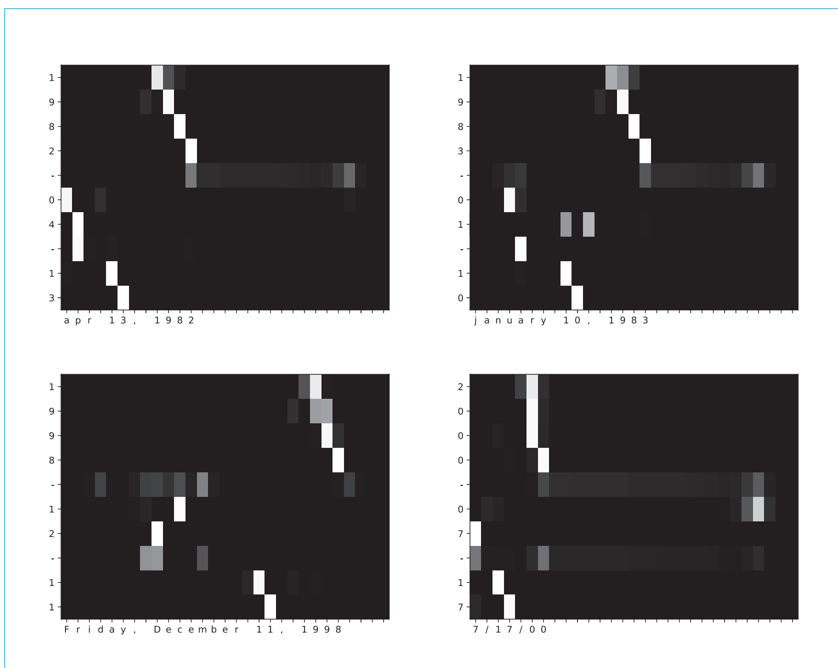


图 8-28　Attention 权重的例子

像这样，使用 Attention，seq2seq 能像我们人一样将注意力集中在必要的信息上。换言之，借助 Attention，我们理解了模型是如何工作的。



我们没有办法理解神经网络内部进行了什么工作（基于何种逻辑工作），而 Attention 赋予了模型"人类可以理解的结构和意义"。在上面的例子中，通过 Attention，我们看到了单词和单词之间的关联性。由此，我们可以判断模型的工作逻辑是否符合人类的逻辑。

以上就是关于 Attention 的评价的内容。通过这里的实验，我们体验了 Attention 的奇妙效果。至此，Attention 的核心话题就要告一段落了，但是关于 Attention 的其他内容还有不少。下一节我们继续围绕 Attention，介绍它的几个高级技巧。

8.4 关于Attention的其他话题

到目前为止，我们研究了 Attention（正确地说，是带 Attention 的 seq2seq），本节我们介绍几个之前未涉及的话题。

8.4.1 双向RNN

这里我们关注 seq2seq 的编码器。首先复习一下，上一节之前的编码器如图 8-29 所示。

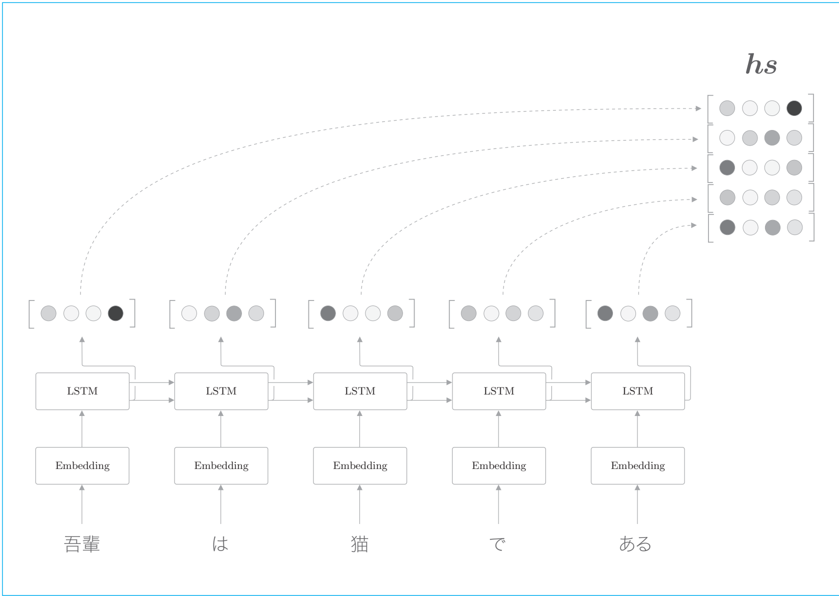


图8-29 基于LSTM层输出 hs

如图 8-29 所示，LSTM 中各个时刻的隐藏状态向量被整合为 $\boldsymbol{hs}$。这里，编码器输出的 $\boldsymbol{hs}$ 的各行中含有许多对应单词的成分。

需要注意的是，我们是从左向右阅读句子的。因此，在图 8-29 中，单词“猫”的对应向量编码了“吾輩”“は”“猫”这 3 个单词的信息。如果考

虑整体的平衡性，我们希望向量能更均衡地包含单词“猫”周围的信息。



在这次的翻译问题中，我们获得了所有的时序数据(需要翻译的文本)。因此，我们既可以从左向右读取文本，也可以从右向左读取文本。

为此，可以让 LSTM 从两个方向进行处理，这就是名为双向 LSTM 的技术，如图 8-30 所示。

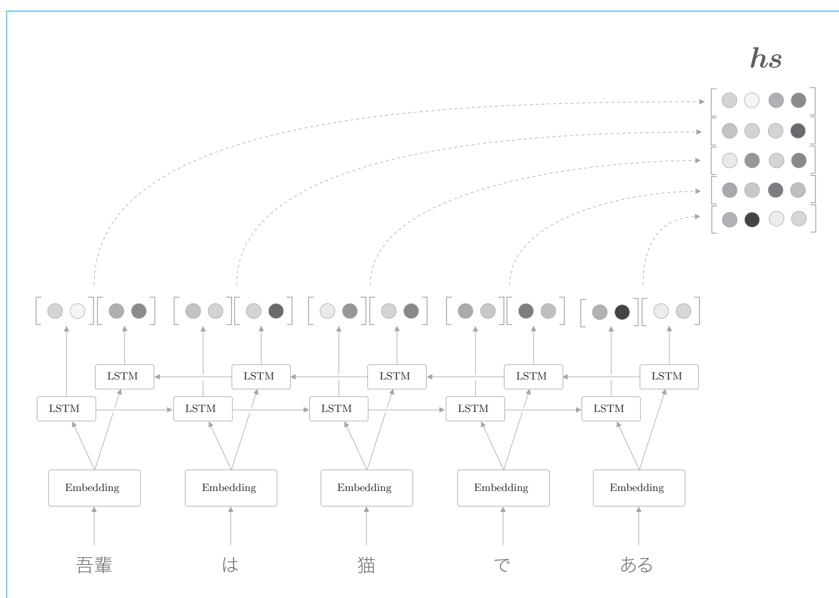


图8-30 基于双向LSTM进行编码的例子(这里简化了LSTM层)

如图 8-30 所示，双向 LSTM 在之前的 LSTM 层上添加了一个反方向处理的 LSTM 层。然后，拼接各个时刻的两个 LSTM 层的隐藏状态，将其作为最后的隐藏状态向量（除了拼接之外，也可以“求和”或者“取平均”等）。

通过这样的双向处理，各个单词对应的隐藏状态向量可以从左右两个方向聚集信息。这样一来，这些向量就编码了更均衡的信息。

双向 LSTM 的实现非常简单。一种实现方式是准备两个 LSTM 层（本章中是 Time LSTM 层），并调整输入各个层的单词的排列。具体而言，其中一个层的输入语句与之前相同，这相当于从左向右处理输入语句的常规的 LSTM 层。而另一个 LSTM 层的输入语句则按照从右到左的顺序输入。如果原文是“A B C D”，就改为“D C B A”。通过输入改变了顺序的输入语句，另一个 LSTM 层从右向左处理输入语句。之后，只需要拼接这两个 LSTM 层的输出，就可以创建双向 LSTM 层。



为了便于理解，本章使用了单向 LSTM 作为编码器。不过，显然也可以将此处介绍的双向 LSTM 用作编码器。感兴趣的读者可以尝试实现使用双向 LSTM 的带 Attention 的 seq2seq。另外，`common/time_layers.py` 的 `TimeBiLSTM` 类中有双向 LSTM 的实现，感兴趣的读者可以参考一下。

8.4.2 Attention 层的使用方法

接下来，我们思考 Attention 层的使用方法。截止到目前，我们使用的 Attention 层的层结构如图 8-31 所示。

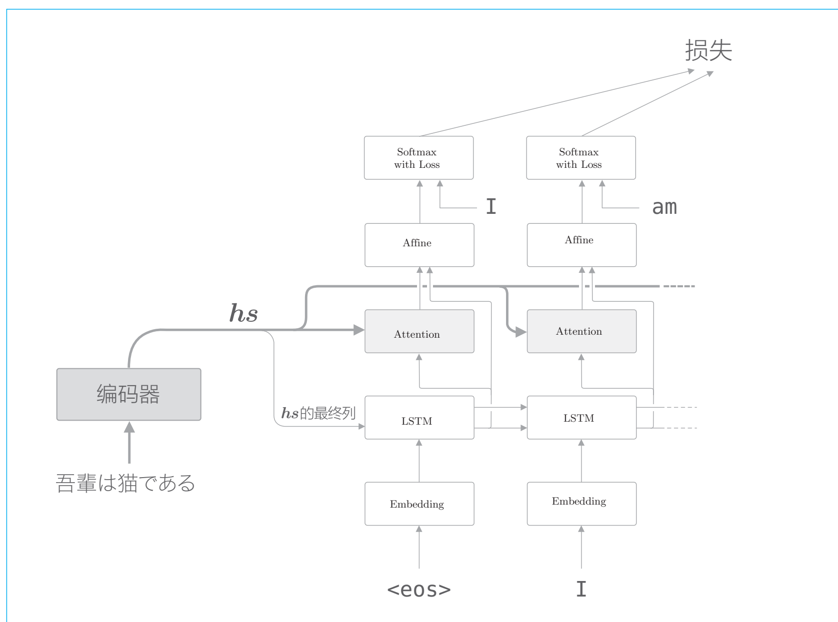


图8-31　上一节之前使用的带Attention的seq2seq的层结构

如图8-31所示，我们将Attention层插入了LSTM层和Affine层之间，不过使用Attention层的方式并不一定非得像图8-31那样。实际上，使用Attention的模型还有其他好几种方式。比如，文献[48]中以图8-32的结构使用了Attention。

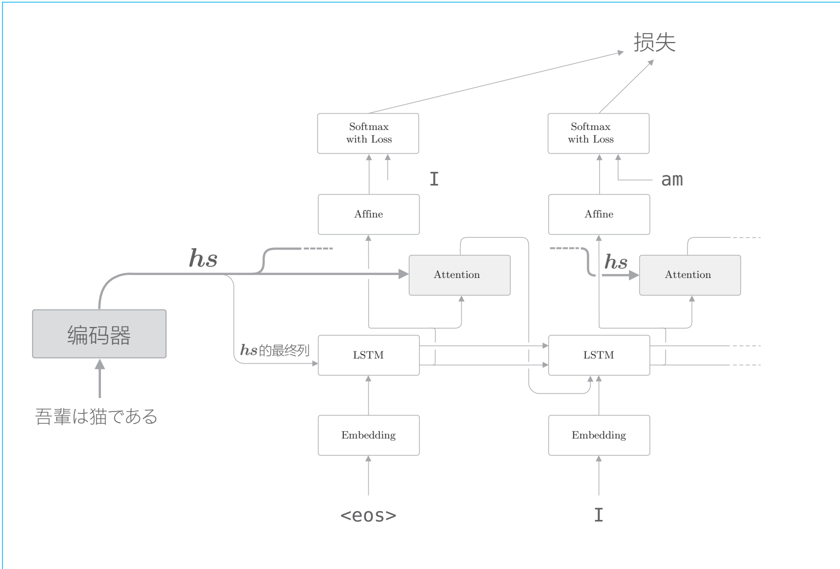


图8-32 Attention层的其他使用示例（这里参考了文献[48]，并简化了网络结构）

在图 8-32 中，Attention 层的输出（上下文向量）被连接到了下一时刻的 LSTM 层的输入处。通过这种结构，LSTM 层得以使用上下文向量的信息。相对地，我们实现的模型则是 Affine 层使用了上下文向量。

那么，Attention 层的位置的不同对最终精度有何影响呢？答案要试一下才知道。实际上，这只能使用真实数据来验证。不过，在上面的两个模型中，上下文向量都得到了很好的应用。因此，在这两个模型之间，我们可能看不到太大的精度差异。

从实现的角度来看，前者的结构（在 LSTM 层和 Affine 层之间插入 Attention 层）更加简单。这是因为在前者的结构中，解码器中的数据是从下往上单向流动的，所以 Attention 层的模块化会更加简单。实际上，我们轻松地将其模块化为了 Time Attention 层。

8.4.3 seq2seq的深层化和skip connection

在诸如翻译这样的实际应用中，需要解决的问题更加复杂。在这种情

况下，我们希望带 Attention 的 seq2seq 具有更强的表现力。此时，首先可以考虑到的是加深 RNN 层（LSTM 层）。通过加深层，可以创建表现力更强的模型，带 Attention 的 seq2seq 也是如此。那么，如果我们加深带 Attention 的 seq2seq，结果会怎样呢？图 8-33 给出了一个例子。

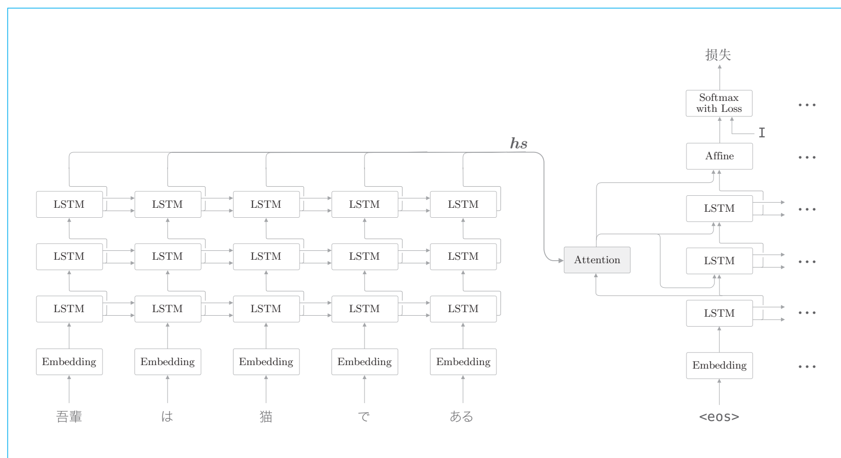


图8-33　使用了3层LSTM层的带Attention的seq2seq

在图 8-33 的模型中，编码器和解码器使用了 3 层 LSTM 层。如本例所示，编码器和解码器中通常使用层数相同的 LSTM 层。另外，Attention 层的使用方法有许多变体。这里将解码器 LSTM 层的隐藏状态输入 Attention 层，然后将上下文向量（Attention 层的输出）传给解码器的多个层（LSTM 层和 Affine 层）。

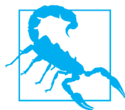


图8-33的模型只是一个例子。除了这个例子之外，还有很多方式，比如使用多个Attention层，或者将Attention的输出输入给下一个时刻的LSTM层等。另外，如上一章所述，在加深层时，避免泛化性能的下降非常重要。此时，Dropout、权重共享等技术可以发挥作用。

另外，在加深层时使用到的另一个重要技巧是残差连接（skip connection，

也称为 residual connection 或 shortcut)。如图 8-34 所示，这是一种“跨层连接”的简单技巧。

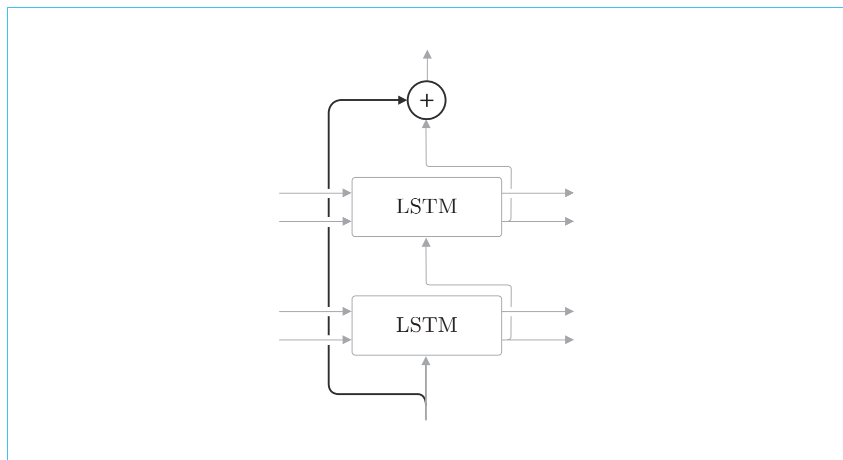


图8-34 LSTM层中的skip connection的例子

如图 8-34 所示，所谓残差连接，就是指“跨层连接”。此时，在残差连接的连接处，有两个输出被相加。请注意这个加法（确切地说，是对应元素的加法）非常重要。因为加法在反向传播时“按原样”传播梯度，所以残差连接中的梯度可以不受任何影响地传播到前一个层。这样一来，即便加深了层，梯度也能正常传播，而不会发生梯度消失（或者梯度爆炸），学习可以顺利进行。



在时间方向上，RNN 层的反向传播会出现梯度消失或梯度爆炸的问题。梯度消失可以通过 LSTM、GRU 等 Gated RNN 应对，梯度爆炸可以通过梯度裁剪应对。而对于深度方向上的梯度消失，这里介绍的残差连接很有效。

8.5 Attention 的应用

到目前为止，我们仅将 Attention 应用在了 seq2seq 上，但是 Attention 这一想法本身是通用的，在应用上还有更多的可能性。实际上，在近些年的深度学习研究中，作为一种重要技巧，Attention 出现在了各种各样的场景中。本节我们将介绍 3 个使用了 Attention 的前沿研究，以使读者感受到 Attention 的重要性和可能性。

8.5.1 GNMT

回看机器翻译的历史，我们可以发现主流方法随着时代的变迁而演变。具体来说，就是从“基于规则的翻译”到“基于用例的翻译”，再到“基于统计的翻译”。现在，神经机器翻译（Neural Machine Translation）取代了这些过往的技术，获得了广泛关注。



神经机器翻译这个术语是出于与之前的基于统计的翻译进行对比而使用的，现在已经成为使用了 seq2seq 的机器翻译的统称。

从 2016 年开始，谷歌翻译就开始将神经机器翻译用于实际的服务，其机器翻译系统称为 GNMT（Google Neural Machine Translation，谷歌神经机器翻译系统）。关于 GNMT 的技术细节，文献 [50] 中有具体介绍。这里，我们以层结构为中心来看一下 GNMT 的架构，如图 8-35 所示。

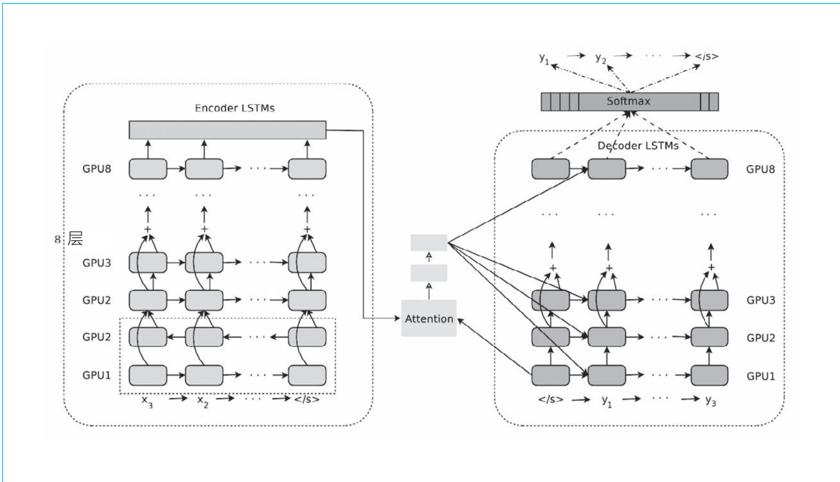


图8-35 GNMT的层结构（引自文献[50]）（参见彩图）

GNMT 和本章实现的带 Attention 的 seq2seq 一样，由编码器、解码器和 Attention 构成。不过，与我们的简单模型不同，这里可以看到许多为了提高翻译精度而做的改进，比如 LSTM 层的多层化、双向 LSTM（仅编码器的第 1 层）和 skip connection 等。另外，为了提高学习速度，还进行了多个 GPU 上的分布式学习。

除了上述在架构上下的功夫之外，GNMT 还进行了低频词处理、用于加速推理的量化（quantization）等工作。利用这些技巧，GNMT 获得了非常好的结果，实际报告出来的结果如图 8-36 所示。

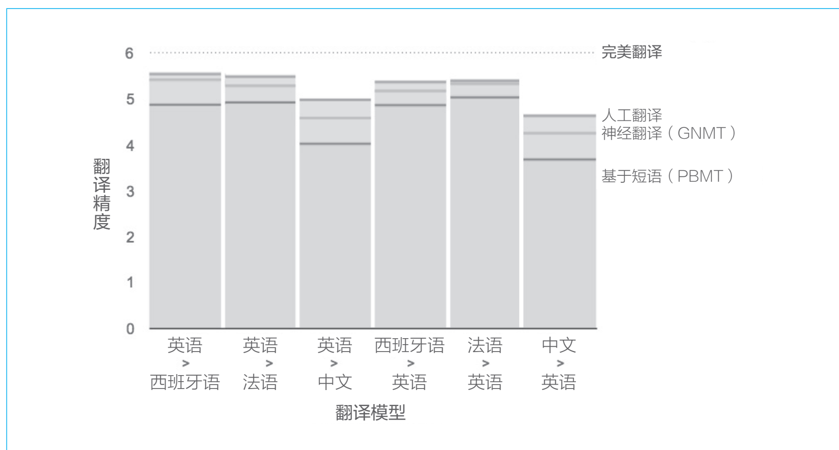


图 8-36 GNMT 的精度评价：纵轴是人按照 0~6 对翻译质量进行的评价（引自文献 [51]）（参见彩图）

如图 8-36 所示，与基于短语的机器翻译（基于统计的机器翻译的一种）这种传统方法相比，GNMT 成功地提高了翻译精度，其精度进一步接近了人工翻译的精度。像这样，GNMT 给出了出色的结果，充分展示了神经翻译的实用性和可能性。不过，但凡用过谷歌翻译的人都知道，它仍存在许多不自然的翻译以及人绝对不会犯的错误。机器翻译的研究仍在继续。实际上，GNMT 只是一个开始，目前围绕神经翻译的研究非常活跃。



实现 GNMT 需要大量的数据和计算资源。根据文献 [50]，GNMT 使用了大量的训练数据，（1 个模型）在将近 100 个 GPU 上学习了 6 天。另外，GNMT 也在设法基于可以并行学习 8 个模型的集成学习和强化学习等技术进一步提高精度。虽然这些事情不是一个人可以完成的，但是我们已经学习了需要用到的技术的核心部分。

8.5.2 Transformer

到目前为止，我们在各种地方使用了 RNN（LSTM）。从语言模型到文本生成，从 seq2se 到带 Attention 的 seq2seq 及其组成部分，RNN 都会出现。使用 RNN 可以很好地处理可变长度的时序数据，（在大多数情况下）

能够获得良好的结果。但是，RNN 也有缺点，比如并行处理的问题。

RNN 需要基于上一个时刻的计算结果逐步进行计算，因此（基本）不可能在时间方向上并行计算 RNN。在使用了 GPU 的并行计算环境下进行深度学习时，这一点会成为很大的瓶颈，于是我们就有了避开 RNN 的动机。

在这样的背景下，现在关于去除 RNN 的研究（可以并行计算的 RNN 的研究）很活跃，其中一个著名的模型是 Transformer[52] 模型。Transformer 是在“Attention is all you need”这篇论文中提出来的方法。如论文标题所示，Transformer 不用 RNN，而用 Attention 进行处理。这里，我们简单地看一下这个 Transformer。



除了 Transformer 之外，还有多个研究致力于去除 RNN，比如用卷积层（Convolution 层）代替 RNN 的研究[54]。这里我们不去探讨该研究的细节，基本上就是用卷积层代替 RNN 来构成 seq2seq，并据此实现并行计算。

Transformer 是基于 Attention 构成的，其中使用了 Self-Attention 技巧，这一点很重要。Self-Attention 直译为“自己对自己的 Attention”，也就是说，这是以一个时序数据为对象的 Attention，旨在观察一个时序数据中每个元素与其他元素的关系。用 Time Attention 层来说明的话，Self-Attention 如图 8-37 所示。

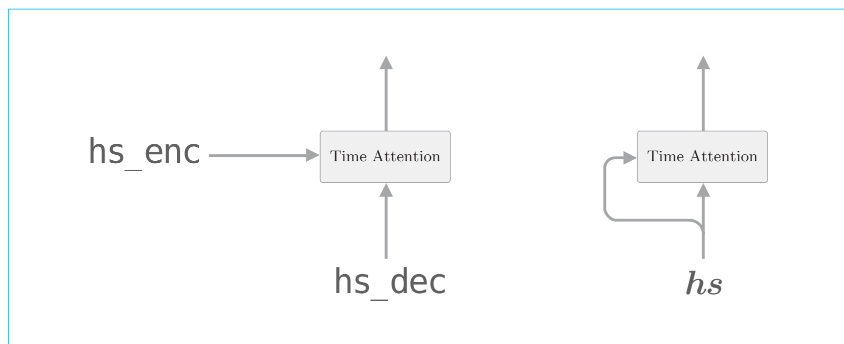


图 8-37 左图是常规的 Attention，右图是 Self-Attention

在此之前，我们用 Attention 求解了翻译这种两个时序数据之间的对应关系。如图 8-37 的左图所示，Time Attention 层的两个输入中输入的是不同的时序数据。与之相对，如图 8-37 的右图所示，Self-Attention 的两个输入中输入的是同一个时序数据。像这样，可以求得一个时序数据内各个元素之间的对应关系。

至此，对 Self-Attention 的说明就结束了，下面我们看一下 Transformer 的层结构，如图 8-38 所示。

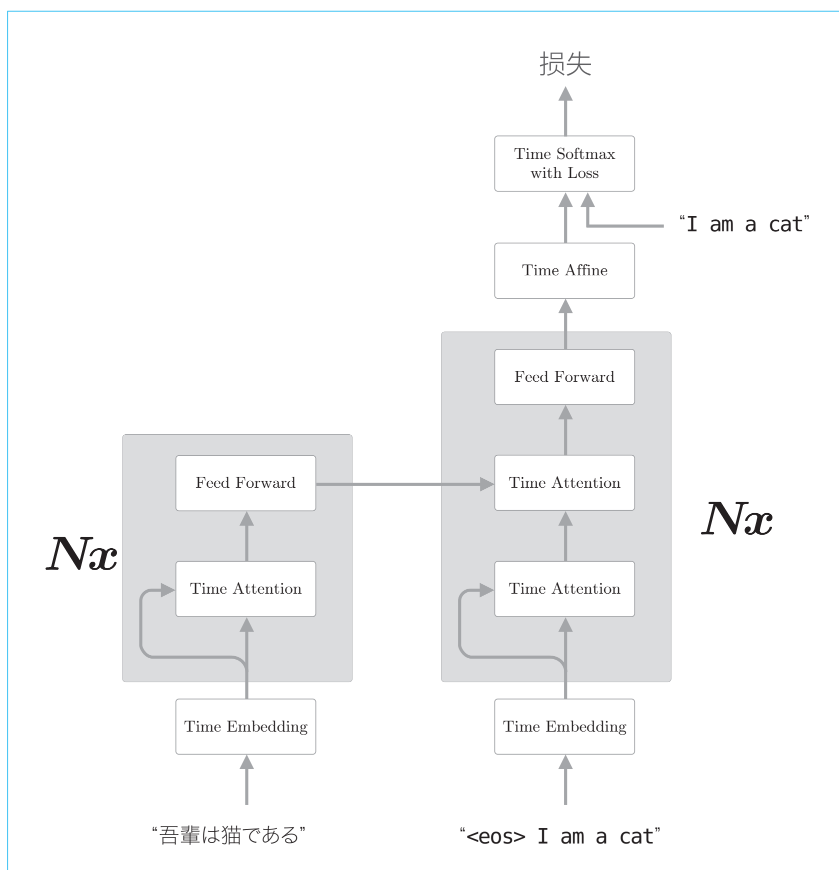


图 8-38　Transformer 的层结构（这里参考了文献 [52]，并简化了模型）

Transformer 中用 Attention 代替了 RNN。实际上，由图 8-38 可知，编码器和解码器两者都使用了 Self-Attention。图 8-38 中的 Feed Forward 层表示前馈神经网络（在时间方向上独立的网络）。具体而言，使用具有一个隐藏层、激活函数为 ReLU 的全连接的神经网络。另外，图中的 ***Nx*** 表示灰色背景包围的元素被堆叠了 N 次。



图 8-38 显示的是简化了的 Transformer。实际上，除了这个架构外，Skip Connection、Layer Normalization[8] 等技巧也会被用到。其他常见的技巧还有，（并行）使用多个 Attention、编码时序数据的位置信息（Positional Encoding，位置编码）等。

使用 Transformer 可以控制计算量，充分利用 GPU 并行计算带来的好处。其结果是，与 GNMT 相比，Transformer 的学习时间得以大幅减少。在翻译精度方面，如图 8-39 所示，也实现了精度提升。

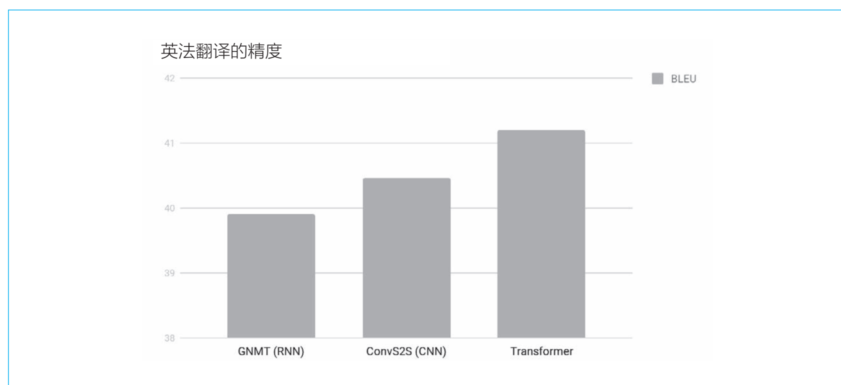


图8-39　使用基准翻译数据 WMT，评价英法翻译的精度。纵轴是翻译精度的指标 BLEU 值，这个值越高越好（参考文献 [53]）

图 8-39 比较了 3 种方法。结果是，使用卷积层的 seq2seq（图中记为 ConvS2S）比 GNMT 精度高，而 Transformer 比使用卷积层的 seq2seq 还要高。如此，不仅仅是计算量，从精度的角度来看，Attention 也是很有前途的技术。

我们之前组合使用了 Attention 和 RNN，但是由这个研究可知，Attention 其实可以用来替换 RNN。这样一来，利用 Attention 的机会可能会进一步增加。

8.5.3 NTM

我们在解决复杂问题时，经常使用纸和笔。从另一个角度来看，这可以解释为基于纸和笔这样的“外部存储装置”，我们的能力获得了延伸。同样地，利用外部存储装置，神经网络也可以获得额外的能力。本节我们讨论的主题就是“基于外部存储装置的扩展”。



RNN和LSTM能够使用内部状态来存储时序数据，但是它们的内部状态长度固定，能塞入其中的信息量有限。因此，可以考虑在RNN的外部配置存储装置（内存），适当地记录必要信息。

在带 Attention 的 seq2seq 中，编码器对输入语句进行编码。然后，解码器通过 Attention 使用被编码的信息。这里需要注意的仍是 Attention 的存在。基于 Attention，编码器和解码器实现了计算机中的“内存操作”。换句话说，这可以解释为，编码器将必要的信息写入内存，解码器从内存中读取必要的信息。

可见计算机的内存操作可以通过神经网络复现。我们可以立刻想到一个方法：在 RNN 的外部配置一个存储信息的存储装置，并使用 Attention 向这个存储装置读写必要的信息。实际上，这样的研究有好几个，NTM（Neural Turing Machine，神经图灵机）[55] 就是其中比较有名的一个。



NTM是DeepMind团队进行的一项研究，后来被改进成名为DNC（Differentiable Neural Computers，可微分神经计算机）[56] 的方法。关于DNC的论文发表在了学术期刊《自然》上。DNC可以认为是强化了内存操作的NTM，但它们的核心技术是一样的。

在解释 NTM 的内容之前，我们先来看一下 NTM 的整体框架。图 8-40

这张有趣的图片非常适合用于这一目的。这是 NTM 所进行的处理的概念表示，很好地总结了 NTM 的精髓（准确地说，这是发展了 NTM 的 DNC 的一篇解说文章 [57] 中用到的图）。

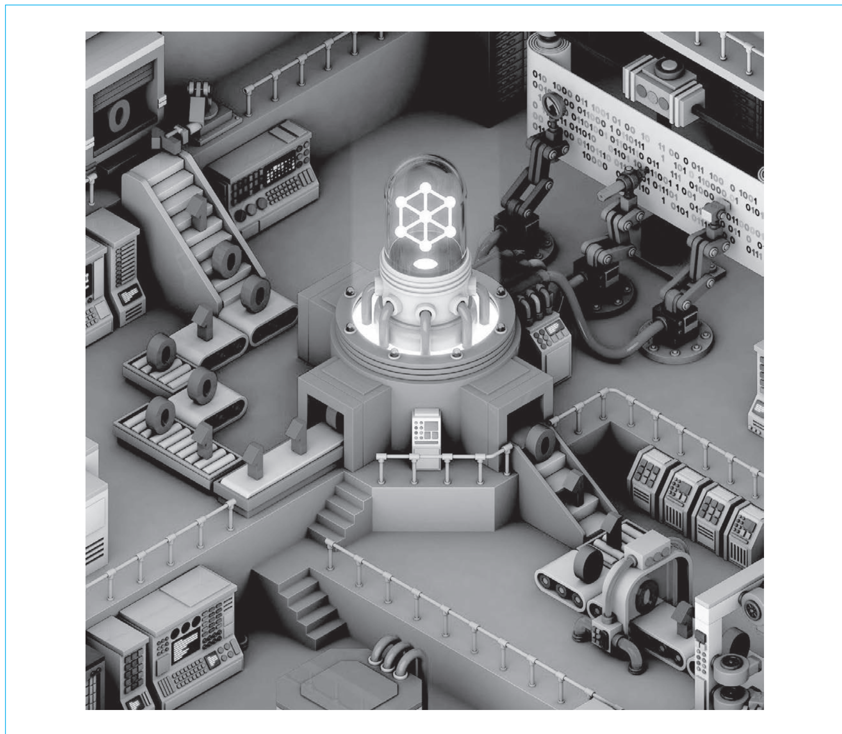


图8-40　NTM的概念图（引自文献[57]）

现在我们看一下图 8-40。这里需要注意的是图中间的一个被称为“控制器”的模块。这是处理信息的模块，我们假定它使用神经网络（或者 RNN）。从图中可以看出，数据“0”和“1”一个接一个地流入这个控制器，控制器对其进行处理并输出新的数据。

这里重要的是，在这个控制器的外侧有一张“大纸”（内存）。基于这个内存，控制器获得了计算机（图灵机）的能力。具体来说，这个能力是指，在这张“大纸”上写入必要的信息、擦除不必要的信息，以及读取必要信息

的能力。顺便说一下，因为图 8-40 的“大纸”是卷式的，所以各个节点可以在需要的地方读写数据。换句话说，就是可以移动到目标地点。

像这样，NTM 在读写外部存储装置的同时处理时序数据。NTM 的有趣之处在于使用“可微分”的计算构建了这些内存操作。因此，它可以从数据中学习内存操作的顺序。



计算机根据人编写的程序进行动作。与此相对，NTM 从数据中学习程序。也就是说，这意味着它可以从“算法的输入和输出”中学习“算法自身”(逻辑)。

NTM 像计算机一样读写外部存储装置，其层结构可以简单地绘制成图 8-41。

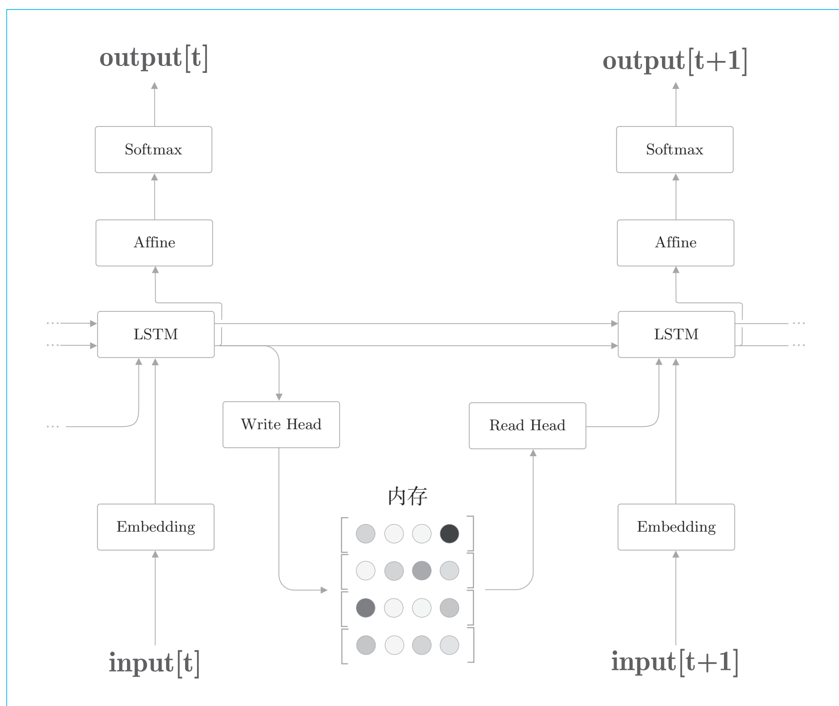


图 8-41　NTM的层结构：新出现了Write Head层和Read Head层，它们进行内存读写

图 8-41 是简化版的 NTM 的层结构。这里 LSTM 层是控制器，执行 NTM 的主要处理。Write Head 层接收 LSTM 层各个时刻的隐藏状态，将必要的信息写入内存。Read Head 层从内存中读取重要信息，并传递给下一个时刻的 LSTM 层。

那么，图 8-41 的 Write Head 层和 Read Head 层如何进行内存操作呢？当然是使用 Attention。



重申一下，在读取（或者写入）内存中某个地址上的数据时，我们需要先“选择”数据。这个选择操作自身是不能微分的，因此先使用 Attention 选择所有地址上的数据，再利用权重表示每个数据的贡献，这样就能够利用可微分的计算替代选择这个操作。

为了模仿计算机的内存操作，NTM 的内存操作使用了两个 Attention，分别是“基于内容的 Attention”和“基于位置的 Attention”。基于内容的 Attention 和我们之前介绍的 Attention 一样，用于从内存中找到某个向量（查询向量）的相似向量。

而基于位置的 Attention 用于从上一个时刻关注的内存地址（内存的各个位置的权重）前后移动。这里我们省略对其技术细节的探讨，具体可以通过一维卷积运算实现。基于内存位置的移动功能，可以再现“一边前进（一个内存地址）一边读取”这种计算机特有的活动。



NTM 的内存操作比较复杂。除了上面说到的操作以外，还包括锐化 Attention 权重的处理、加上上一个时刻的 Attention 权重的处理等。

通过自由地使用外部存储装置，NTM 获得了强大的能力。实际上，对于 seq2seq 无法解决的复杂问题，NTM 取得了惊人的成绩。具体而言，NTM 成功解决了长时序的记忆问题、排序问题（从大到小排列数字）等。

如此，NTM 借助外部存储装置获得了学习算法的能力，其中 Attention 作为一项重要技术而得到了应用。基于外部存储装置的扩展技术和 Attention

会越来越重要，今后将被应用在各种地方。

8.6 小结

本章我们学习了 Attention 的结构，并实现了 Attention 层。然后，我们使用 Attention 实现了 seq2seq，并通过简单的实验，确认了 Attention 的出色效果。另外，我们对模型推理时的 Attention 的权重（概率）进行了可视化。从结果可知，具有 Attention 的模型以与人类相同的方式将注意力放在了必要的信息上。

另外，本章还介绍了有关 Attention 的前沿研究。从多个例子可知，Attention 扩展了深度学习的可能性。Attention 是一种非常有效的技术，具有很大潜力。在深度学习领域，今后 Attention 自己也将吸引更多的“注意力”。

本章所学的内容

- 在翻译、语音识别等将一个时序数据转换为另一个时序数据的任务中，时序数据之间常常存在对应关系
- Attention 从数据中学习两个时序数据之间的对应关系
- Attention 使用向量内积（方法之一）计算向量之间的相似度，并输出这个相似度的加权和向量
- 因为 Attention 中使用的运算是可微分的，所以可以基于误差反向传播法进行学习
- 通过将 Attention 计算出的权重（概率）可视化，可以观察输入与输出之间的对应关系
- 在基于外部存储装置扩展神经网络的研究示例中，Attention 被用来读写内存

附录 A sigmoid 函数和 tanh 函数的导数

神经网络使用各种各样的激活函数，这里将讨论具有代表性的 sigmoid 函数和 tanh 函数，使用两种不同的方法计算这两个函数的导数。具体来说，就是使用计算图计算 sigmoid 函数的导数，使用数学式计算 tanh 函数的导数。通过理解各个方法，来熟悉导数的计算。

A.1 sigmoid 函数

sigmoid 函数可用下式表示：

$$y = \frac{1}{1 + \exp(-x)} \tag{A.1}$$

此时，式 (A.1) 的计算图如图 A-1 所示。

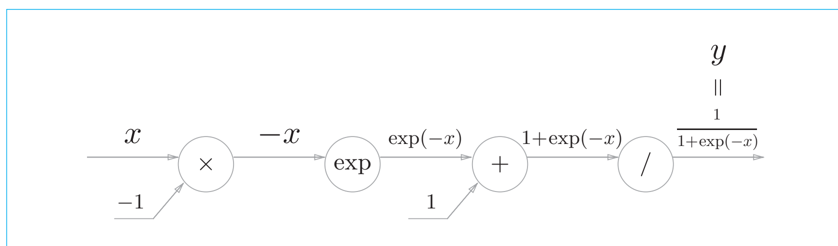


图 A-1 Sigmoid 层的计算图

在图 A-1 中，除了“×”和“+”节点之外，还有 exp 和“/”节点。exp 节点进行 $y=\exp(x)$ 的计算，“/”节点进行 $y=\frac{1}{x}$ 的计算。现在，我们使用计算图，来逐一确认它们的反向传播。

步骤 1

“/”节点表示 $y=\frac{1}{x}$，它的导数可以解析性地如下表示：

$$\frac{\partial y}{\partial x}=-\frac{1}{x^2}=-\left(\frac{1}{x}\right)^2=-y^2 \tag{A.2}$$

基于式 (A.2)，在反向传播时，对上游的梯度乘以 $-y^2$（正向传播的输出的平方取负），再传递给下游，如图 A-2 所示。

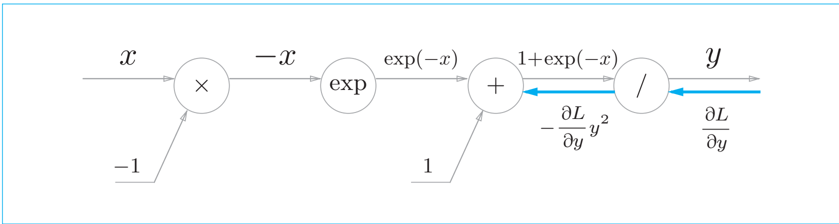


图 A-2　反向传播的步骤 1

步骤 2

“+”节点将上游的值原样传递给下游，它的计算图如图 A-3 所示。

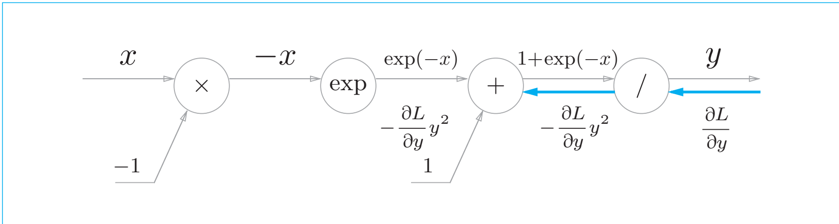


图 A-3　反向传播的步骤 2

步骤 3

exp 节点表示 $y = \exp(x)$，它的导数可用下式表示：

$$\frac{\partial y}{\partial x} = \exp(x) \tag{A.3}$$

在计算图中，对上游的梯度乘以正向传播时的输出（此处为 $\exp(-x)$），再传递给下游（图 A-4）。

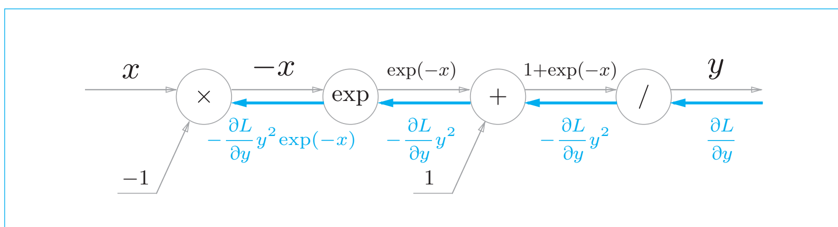


图 A-4　反向传播的步骤 3

步骤 4

“×”节点乘以将正向传播时的输入交换后的值，这里乘以 -1（图 A-5）。

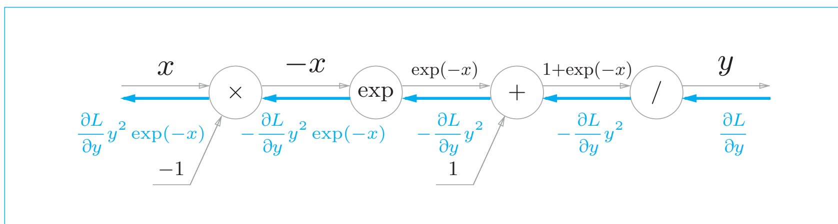


图 A-5　sigmoid 函数的计算图（反向传播）

如上所示，Sigmoid 层可以按图 A-5 的计算图进行反向传播。从图 A-5 的结果可知，反向传播的输出是 $\frac{\partial L}{\partial y}y^2\exp(-x)$，这个值被传递给下游节点。这个 $\frac{\partial L}{\partial y}y^2\exp(-x)$ 可以进一步整理如下：

$$
\begin{aligned}
\frac{\partial L}{\partial y} y^2 \exp(-x) &= \frac{\partial L}{\partial y} \frac{1}{(1+\exp(-x))^2} \exp(-x) \\
&= \frac{\partial L}{\partial y} \frac{1}{1+\exp(-x)} \frac{\exp(-x)}{1+\exp(-x)} \\
&= \frac{\partial L}{\partial y} y(1-y)
\end{aligned}
\tag{A.4}
$$

由式 (A.4) 的展开可知，仅根据正向传播时的输出，就可以计算 sigmoid 函数的反向传播。综上所述，sigmoid 函数的计算图可以画成图 A-6。

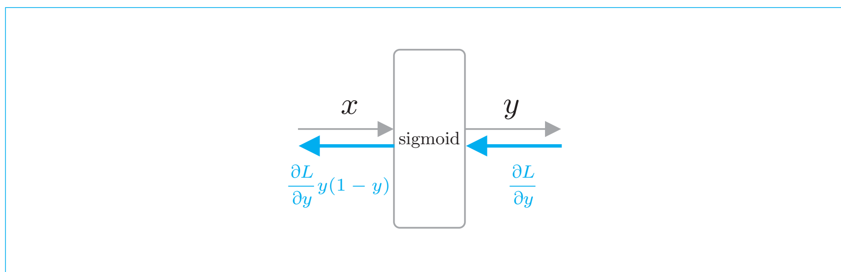


图 A-6 sigmoid 函数的计算图

以上就是 sigmoid 函数的导数。这里使用计算图计算了 sigmoid 函数的导数。下面来看一下如何解析性地计算 tanh 函数的导数。

A.2 tanh 函数

tanh 函数也称为双曲正切（hyperbolic tangent）函数，可以用式 (A.5) 表示：

$$
y = \tanh(x) = \frac{\mathrm{e}^x - \mathrm{e}^{-x}}{\mathrm{e}^x + \mathrm{e}^{-x}} \tag{A.5}
$$

我们的目标是对式 (A.5) 求 $\frac{\partial y}{\partial x}$。因此，这里使用下面的导数公式：

$$\left\{\frac{f(x)}{g(x)}\right\}' = \frac{f'(x)g(x) - f(x)g'(x)}{g(x)^2} \tag{A.6}$$

式 (A.6) 是分数函数的导数公式。为了便于观察，这里将 $\frac{f(x)}{g(x)}$ 记为 $\left\{\frac{f(x)}{g(x)}\right\}'$。同样地，将 $f(x)$ 关于 x 的导数记为 $f'(x)$。

另外，关于纳皮尔数（e），可以解析性地推导出下面的导数：

$$\frac{\partial \mathrm{e}^x}{\partial x} = \mathrm{e}^x \tag{A.7}$$

$$\frac{\partial \mathrm{e}^{-x}}{\partial x} = -\mathrm{e}^{-x} \tag{A.8}$$

通过使用上面的式 (A.6)、式 (A.7) 和式 (A.8)，tanh 函数的导数可以如下求出：

$$\begin{aligned}
\frac{\partial \tanh(x)}{\partial x} &= \frac{(\mathrm{e}^x + \mathrm{e}^{-x})(\mathrm{e}^x + \mathrm{e}^{-x}) - (\mathrm{e}^x - \mathrm{e}^{-x})(\mathrm{e}^x - \mathrm{e}^{-x})}{(\mathrm{e}^x + \mathrm{e}^{-x})^2} \\
&= 1 - \frac{(\mathrm{e}^x - \mathrm{e}^{-x})(\mathrm{e}^x - \mathrm{e}^{-x})}{(\mathrm{e}^x + \mathrm{e}^{-x})^2} \\
&= 1 - \left\{\frac{(\mathrm{e}^x - \mathrm{e}^{-x})}{(\mathrm{e}^x + \mathrm{e}^{-x})}\right\}^2 \\
&= 1 - \tanh(x)^2 \\
&= 1 - y^2
\end{aligned} \tag{A.9}$$

如式 (A.9) 所示，使用分数函数的导数公式，并对公式进行简单的整理，就可以求出 tanh 函数的导数，结果是 $1 - y^2$。根据这个结果，tanh 函数的计算图可以绘制成图 A-7。

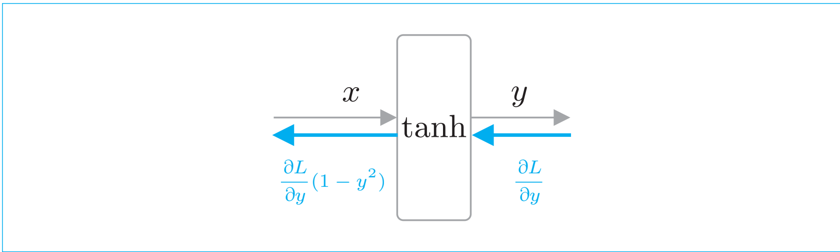


图 A-7 tanh 函数的计算图

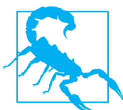
以上就是 tanh 函数的导数的推导过程。我们通过展开数学式，简洁明了地求得了导数。

A.3 小结

以上我们分别用计算图和解析性的方法求解了导数。两种方法殊途同归，读者可以根据不同的问题选择合适的方法。不过，在习惯之后，可能会觉得使用数学式的方法更加方便。而在刚开始时，可能计算图这种直观的方法比较好（特别是心中有疑惑时）。能用多个方法解决问题非常重要！就像本附录这样，在解决问题时，有时使用数学式，有时使用计算图，这可以加深我们的理解。

附录B 运行WordNet

在本附录中，我们将实际运行一下 WordNet，来看一下 WordNet 中存在什么样的“知识”。另外，这里的实现只是为了让读者对该同义词库有所了解，所以进行的实验非常简单。



本附录将简单介绍 WordNet 和 NLTK。《自然语言处理入门》[14]一书中有关于 NLTK 的详细说明，感兴趣的读者可以参考一下。

B.1 NLTK的安装

通过 Python 利用 WordNet，可以使用 NLTK（Natural Language Toolkit，自然语言处理工具包）这个库。NLTK 是用于自然语言处理的 Python 库，其中包含许多自然语言处理相关的便捷功能，比如词性标注、句法分析、信息抽取和语义分析等。

现在我们就来安装 NLTK。安装方法有很多，这里介绍如何使用 pip 进行安装（其他的安装方法，请读者根据自身环境自行尝试）。

要安装 NLTK，需要向终端输入下面一行代码。

```
$ pip install nltk
```

这样 NLTK 的安装就结束了（安装需要一些时间）。在安装结束后，导入 NLTK，以确认安装是否成功。

```
>>> import nltk
>>>
```

这里，启动 Python 解释器，导入 NLTK。如果 NLTK 安装正确，则如上所示，不显示任何错误。

B.2　使用WordNet获得同义词

下面，我们来实际使用一下 WordNet。首先，从 `nltk.corpus` 中导入 `wordnet` 模块。

```
>>> from nltk.corpus import wordnet
```

准备完毕。我们试着查看一下单词 car 的同义词，在此之前，我们先看一下单词 car 存在多少个不同的含义。为此，使用 `wordnet.synsets()` 方法。



在 WordNet 中，每个单词都被归类到了名为 synset 的同义词簇中。为了得到 car 的同义词簇，只需要调用 `wordnet.synset()` 方法。这里有一点需要注意，那就是单词 car（和其他许多单词一样）存在多个含义。具体来说，除了“汽车”的含义之外，它还有“（火车）车厢”“吊舱”的含义。因此，在获得同义词时，需要（从多个含义中）指定是哪个含义。

现在，我们尝试用 WordNet 获得 car 的同义词。

```
>>> wordnet.synsets('car')
[Synset('car.n.01'),
 Synset('car.n.02'),
 Synset('car.n.03'),
 Synset('car.n.04'),
 Synset('cable_car.n.01')]
```

这里输出了一个包含 5 个元素的列表，这表示 car 这个单词被定义了 5 种不同的含义（严格来说，是 5 个不同的同义词簇）。

还有一点需要注意的是，上面的列表中的元素显示的是 car 的“标题词”。如图 B-1 所示，WordNet 中使用的标题词由被“.”切分的 3 个元素指定。比如，“car.n.01”这个标题词表示“car 的第 1 个名词”的含义（簇）。

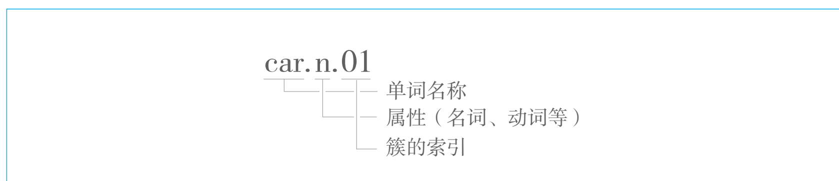


图 B-1　WordNet 中的标题词的解读方法：n 是 noun（名词）的首字母



许多单词都有多个含义。WordNet 中使用标题词从单词的多个含义中指定某个特定的含义。因此，一般情况下，在 WordNet 的方法中需要给参数指定单词名称时，不是指定“car”，而是指定“car.n.01”或者“car.n.02”这样的标题词。

下面，我们来确认“car.n.01”这一标题词指定的同义词的含义。为此，使用 wordnet.synset() 方法，获取“car.n.01”的同义词簇。另外，对该同义词簇调用 definition() 方法。

```
>>> car = wordnet.synset('car.n.01')  # 同义词簇
>>> car.definition()
'a motor vehicle with four wheels; usually propelled by an internal
combustion engine'
```

上面的结果可直译为“使用内燃机驱动的有 4 个车轮的机动车”，这就是“car.n.01”这个同义词簇的含义。另外，这里使用的 definition() 方法主要是用来帮助人（而非计算机）理解该单词的。

现在我们来实际看一下“car.n.01”这个标题词的同义词簇中有什么样的单词。为此，使用 lemma_names() 方法。

```
>>> car.lemma_names()
['car', 'auto', 'automobile', 'machine', 'motorcar']
```

如此，使用 `lemma_names()` 方法，可以获得同义词簇中存在的单词名称。从上面的结果可知，在 car 这个单词（严格地讲，是“car.n.01”这个标题词）中，有 auto、automobile、machine 和 motorcar 这 4 个单词被定义为了同义词。

B.3　WordNet和单词网络

接下来，我们使用 car 的单词网络，查看一下它和其他单词在语义上的上下位关系。为此，可以使用 `hypernym_paths()` 方法。hypernym 主要是语言学中用到的单词，意思是“上位词”。

```
>>> car.hypernym_paths()[0]
[Synset('entity.n.01'), Synset('physical_entity.n.01'),
Synset('object.n.01'), Synset('whole.n.02'), Synset('artifact.n.01'),
Synset('instrumentality.n.03'), Synset('container.n.01'),
Synset('wheeled_vehicle.n.01'), Synset('self-propelled_vehicle.n.01'),
Synset('motor_vehicle.n.01'), Synset('car.n.01')]
```

从上面的结果可知，car 这个单词从 entity 这个单词出发，经过了“entity → physical_entity → object → ··· → motor_vehicle → car”这一路径（此处省略了“标题词”的标记）。这里具体看一下各个单词，car 的上一层是 motor vehicle（机动车），再上一层是 self-propelled vehicle（自走式车辆），继续往上是 object、entity 等抽象单词。在构成 WordNet 的单词网络中，各个单词被配置成越往上走越抽象，越往下走越具体。



在上面的例子中，`car.hypernym_paths()` 返回列表，该列表元素中包含了具体的路径信息。为什么要返回列表呢？因为单词之间的路径可能存在多个。就上面的例子来说，从起点单词 entity 到终点单词 car 有多条路径（有的单词可能只有一条路径）。

B.4　基于WordNet的语义相似度

如前所述，WordNet 中许多单词根据同义词（近义词）被分组。另外，单词之间被构建了语义网络。这些单词之间的关联知识可以应用在很多问题中。这里，我们来看一个计算单词之间的相似度的例子。

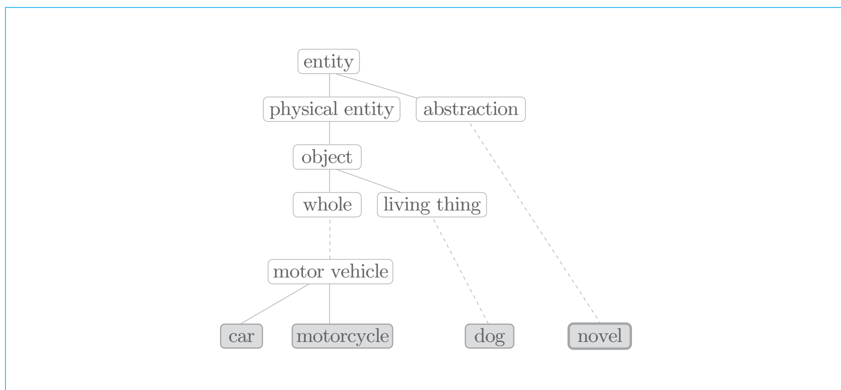
求单词之间的相似度，可以使用 path_similarity() 方法。这个方法返回单词之间的相似度，其返回值是 0 ~ 1 的实数（数值越大，越相似）。现在我们实际地求一下单词之间的相似度。这里分别求单词 car（汽车）和 novel（小说）、dog（狗）、motorcycle（摩托车）之间的相似度。

```
>>> car = wordnet.synset('car.n.01')
>>> novel = wordnet.synset('novel.n.01')
>>> dog = wordnet.synset('dog.n.01')
>>> motorcycle = wordnet.synset('motorcycle.n.01')

>>> car.path_similarity(novel)
0.05555555555555555
>>> car.path_similarity(dog)
0.07692307692307693
>>> car.path_similarity(motorcycle)
0.3333333333333333
```

从上面的结果可知，对于单词 car，单词 motorcycle 的相似度最高，其次是 dog，最不相似的单词是 novel。从相似度的值来看，car 和 motorcycle 的相似度很大，值比其他两个单词大很多倍。这些结果可以说很接近我们的感觉。

上例中用到的 path_similarity() 方法会在内部基于图 B-2 所示的单词网络的公共路径计算单词之间的相似度。



图B-2 基于单词网络的公共路径计算单词的语义相似度（虚线表示途中还有多个单词）

图 B-2 展示了 WordNet 的部分单词网络（省略了途中的单词）。从该图可知，car 和 motorcycle 之间有许多公共路径。实际上，它们到倒数（从底部数）第二个单词 motor vehicle 的路径是相同的。比较 car 和 dog 可以发现，它们的路径在单词 object 处分叉了。再比较 car 和 novel 可以发现，它们的路径在最上面的单词 entity 处就已经分叉了。`path_similarity()` 方法基于这些信息计算单词之间的相似度，（在本例中）其结果很接近我们的感觉。

像这样，使用单词网络，可以计算两个单词之间的相似度。计算单词之间的相似度意味着我们可以测量单词和单词在语义上的距离。而如果没有理解单词含义，就没有办法正确完成此类任务。从这一点来看，同义词库可以说（间接）赋予了计算机理解单词含义的能力。



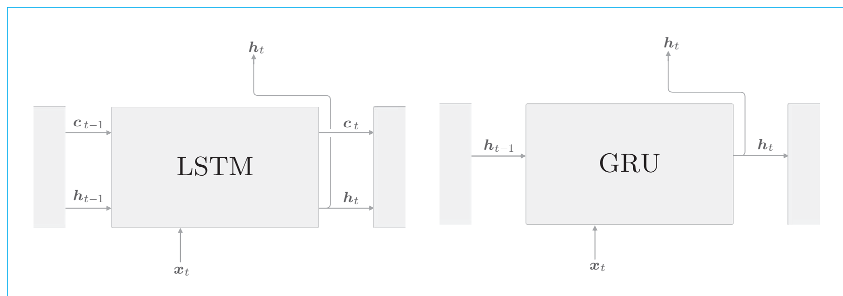
除了`path_similarity()`方法之外，WordNet中还提供了几个用来测量相似度的方法（比如Leacock-Chodorow相似度、Wu-Palmer相似度等）。感兴趣的读者请参考WordNet的Web文档[18]。

附录 C GRU

第 6 章详细介绍了 Gated RNN 的 LSTM。LSTM 是一个非常好的层，但是它的参数太多，计算需要很长时间。因此，最近业界又提出了很多用来替代 LSTM 的 Gated RNN。这里，我们介绍一下 GRU（Gated Recurrent Unit，门控循环单元）[42] 这个有名的 Gated RNN。GRU 保留了 LSTM 使用门的理念，但是减少了参数，缩短了计算时间。现在，我们来看一下 GRU 的内部结构。

C.1 GRU 的接口

LSTM 的重点是使用门，因此学习时梯度的流动平稳，梯度消失得以缓解。GRU 也继承了这一想法。不过，LSTM 和 GRU 存在几个差异，主要区别在于层的接口，如图 C-1 所示。



图C-1　LSTM和GRU的比较

如图 C-1 所示，相对于 LSTM 使用隐藏状态和记忆单元两条线，GRU 只使用隐藏状态。顺便说一下，这和第 5 章讨论的“简单 RNN”的接口相同。



LSTM 的记忆单元是私有存储，对其他层不可见。LSTM 将必要信息记录在记忆单元中，并基于记忆单元的信息计算隐藏状态。与此相对，GRU 中不需要记忆单元这样的额外存储。

C.2　GRU的计算图

现在，我们看一下 GRU 内部进行的计算。这里用数学式表示 GRU 中进行的计算，并给出与之对应的计算图。另外，计算图使用在第 6 章的 LSTM 的计算图中使用的 σ 和 tanh 等简化版节点。

$$\boldsymbol{z} = \sigma(\boldsymbol{x}_t \boldsymbol{W}_x^{(\mathrm{z})} + \boldsymbol{h}_{t-1} \boldsymbol{W}_h^{(\mathrm{z})} + \boldsymbol{b}^{(\mathrm{z})}) \tag{C.1}$$

$$\boldsymbol{r} = \sigma(\boldsymbol{x}_t \boldsymbol{W}_x^{(\mathrm{r})} + \boldsymbol{h}_{t-1} \boldsymbol{W}_h^{(\mathrm{r})} + \boldsymbol{b}^{(\mathrm{r})}) \tag{C.2}$$

$$\tilde{\boldsymbol{h}} = \tanh(\boldsymbol{x}_t \boldsymbol{W}_x + (\boldsymbol{r} \odot \boldsymbol{h}_{t-1}) \boldsymbol{W}_h + \boldsymbol{b}) \tag{C.3}$$

$$\boldsymbol{h}_t = (1 - \boldsymbol{z}) \odot \boldsymbol{h}_{t-1} + \boldsymbol{z} \odot \tilde{\boldsymbol{h}} \tag{C.4}$$

GRU 中进行的计算由上述 4 个式子表示（这里 $\boldsymbol{x}_t$ 和 $\boldsymbol{h}_{t-1}$ 都是行向量），对应的计算图如图 C-2 所示。

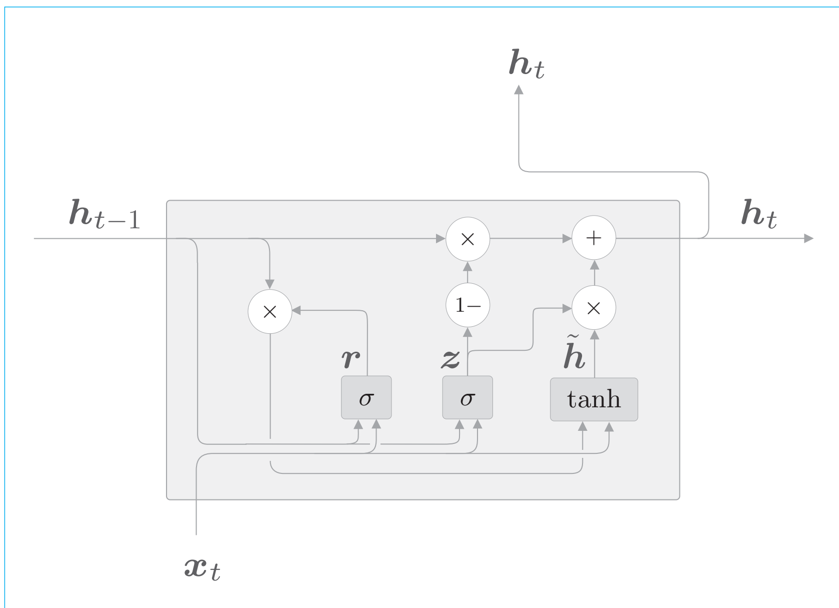


图 C-2　GRU 的计算图：σ 节点和 tanh 节点有专用的权重，节点内部进行仿射变换（“1−”节点输入 x，输出 $1-x$）

如图 C-2 所示，GRU 没有记忆单元，只有一个隐藏状态 $\boldsymbol{h}$ 在时间方向上传播。这里使用 $\boldsymbol{r}$ 和 $\boldsymbol{z}$ 共两个门（LSTM 使用 3 个门），$\boldsymbol{r}$ 称为 reset 门，$\boldsymbol{z}$ 称为 update 门。

$\boldsymbol{r}$（reset 门）决定在多大程度上“忽略”过去的隐藏状态。根据式 (C.3)，如果 $\boldsymbol{r}$ 是 0，则新的隐藏状态 $\tilde{\boldsymbol{h}}$ 仅取决于输入 $\boldsymbol{x}_t$。也就是说，此时过去的隐藏状态将完全被忽略。

而 update 门是更新隐藏状态的门，它扮演了 LSTM 的 forget 门和 input 门两个角色。式 (C.4) 的 $(1-\boldsymbol{z})\odot\boldsymbol{h}_{t-1}$ 部分充当 forget 门的功能。根据这个计算，从过去的隐藏状态中删除应该被遗忘的信息。$\boldsymbol{z}\odot\tilde{\boldsymbol{h}}$ 的部分充当 input 门的功能，对新增的信息进行加权。

综上，GRU 是简化了 LSTM 的架构，与 LSTM 相比，可以减少计算成本和参数。这里，我们不进行 GRU 层的实现，它的代码在 `common/time_layers.py` 中，感兴趣的读者可以参考一下。



那么，我们应该使用LSTM和 GRU 中的哪一个呢？由文献 [32] 和文献 [33] 可知，根据不同的任务和超参数设置，结论可能不同。在最近的研究中，LSTM（以及LSTM的变体）被大量使用，而GRU的人气也在稳步上升。因为GRU的超参数少、计算量小，所以特别适合用于数据集较小、设计模型需要反复实验的场景。

后记

为了登上险峰，一开始要走得慢一点。

——莎士比亚

至此，我们在深度学习的世界中探索了自然语言处理这一主题。实际上，我们实现了关于自然语言处理的各种代码，并通过大量实验学习了若干个重要技术，希望读者能够从这些经历中学到一些东西。如果读者感受到了其中的乐趣与深奥，那笔者将感到无比开心。

现在，深度学习领域正在加速发展，几乎每天都有大量论文发表，新的想法接连不断地被提出来。遗憾的是，我们没有办法把所有论文都看一遍。今后深度学习将如何发展，谁也无法准确预测。

此外，深度学习中的重要技术（在某种程度上）正在固定下来。我相信，本书介绍的围绕自然语言处理的技术今后仍将十分重要。希望读者能够立足于从本书学到的知识，在更广阔的深度学习世界里走得更扎实。

本书到此结束。非常感谢大家阅读本书。从历史的角度来看，我们这一代目睹了深度学习如何一步一步地渗入世界，并改变世界。笔者只是幸运地出生在这一时代，并恰巧写了关于这一主题的书。很高兴有此缘分让笔者通过本书和大家进行了交流。十分感谢！

致谢

本书的存在离不开伟大的先驱们对深度学习、人工智能和自然科学的推动。首先，笔者想感谢他们。其次，还要感谢身边朋友的支持和帮助。他们直接或者间接地鼓励、支持着我。谢谢！

作为一个新的尝试，本书使用了“公开审稿”的校稿方式。在这次公开审稿中，原稿被公开在 Web 上，任何人都可以阅读和评论。结果，仅一个月的审稿时间，我们就收到了 1500 多条有用的评论。感谢所有参加审稿的朋友！毫无疑问，正是大家的评论让本书得到了进一步的完善。谢谢大家！当然，本书中存在的不完善之处或者错误，均是笔者的责任，与评论者无关。

最后，本书的出版还要归功于世界各地人民。笔者每天都在受到许多不知道名字的人的影响，缺少其中任何一个人，本书可能就不存在了（至少不是现在这种状态）。这些话也同样可以对周围的自然界说，河流、树木、大地、天空中都有我的日常生活。感谢这些无名的人，无名的自然。

斋藤康毅

2018 年 6 月 1 日

审稿

寄田明宏	玉川晃之	吉永彰成	高山笃史	佐藤太树
小林大将	高屋英知	江崎大嗣	兵头冲	铃木骏
藤冈秀明	田岛英朗	原田秀隆	河村英幸	谷冈广树
荒井浩	布施快	渊上纮行	濑户口久雄	藤田勇
石川敬规	上久保景幸	神户宏之	金泽隆史	永田员广
石津和纪	古贺一德	佐佐木知哉	大滝启介	山内健太
小泽学	寺田学	长谷川正彦	山本正喜	藤井昌纪
水谷穰	三浦康幸	野口宗之	内藤亮介	米泽直记
佐藤尚至	野口聪明	伊藤宣博	山下修	佐藤隆佑
佐藤温	佐藤洁	新谷正领	小林茂	清水丰
永田晋介	杉田臣辅	和田信也	白木宏朋	中村翔
千田翔太	藤原秀平	铃木英太	阿部考志	宫阪健夫
若杉武史	铃木琢也	新村拓也	田中智史	蔡天星
石川哲朗	鹰城彻	森原利之	大野翼	土井泰法
西田泰士	高野泰朋	长谷部阳一郎	荻原义宽	林悠大
增宫雄一	小池祐二	飞永由夏	宵勇树	藤本裕介
古川悠介	西口祐介	吉成祐人	水原悠	
TAICHI KAWABATA		AIMY——山形县人工智能协会		

制作

武藤健志

增子萌

编辑

宫川直树

岩佐未央

小柳彩良

参考文献

Python 相关

[1] Broadcasting.

[2] 100 numpy exercises.

[3] CuPy web page.

[4] CuPy install page.

深度学习基础

[5] 斎藤康毅 . ゼロから作る Deep Learning —— Python で学ぶディープラーニングの理論と実装 [M]. 東京：オライリー・ジャパン，2016.

[6] Gupta, Suyog, et al. Deep learning with limited numerical precision[J]. Proceedings of the 32nd International Conference on Machine Learning (ICML-15), 2015.

[7] Jouppi, Norman P., et al. In-datacenter performance analysis of a tensor processing unit[J]. Proceedings of the 44th Annual International Symposium on Computer Architecture. ACM, 2017.

[8] Ba, Jimmy Lei, Jamie Ryan Kiros, Geoffrey E. Hinton.Layer Normalization[J]. arXiv preprint arXiv: 1607.06450, 2016.

[9] Srivastava, Nitish, et al. Dropout: a simple way to prevent neural

networks from overfitting[J]. Journal of machine learning research, 2014, 15(1): 1929-1958.

基于深度学习的自然语言处理

[10] Stanford University CS224d: Deep Learning for Natural Language Processing.

[11] Oxford Deep NLP 2017 course.

[12] Young, D. Hazarika, S. Poria, E. Cambria.Recent trends in deep learning based natural language processing[J]. in arXiv preprint arXiv: 1708.02709, 2017.

[13] 坪井祐太，海野裕也，鈴木潤 . 深層学習による自然言語処理（機械学習プロフェッショナルシリーズ）[M]. 東京：講談社，2017.

深度学习出现之前的自然语言处理

[14] Steven Bird, Ewane Klein, Edward Loper. 入門自然言語処理 [M]. 萩原正人，中山敬広，水野貴明，訳. 東京：オライリー・ジャパン，2010.

[15] Jeffrey E. F. Friedl. 詳説正規表現第 3 版 [M]. 株式会社ロングテール，長尾高弘，訳. 東京：オライリー・ジャパン，2008.

[16] Christopher D. Manning, Hinrich Schütze. 統計的自然言語処理の基礎 [M]. 加藤恒昭，菊井玄一郎，林良彦，森辰則，訳. 東京：共立出版，2017.

[17] Miller, George A. WordNet: a lexical database for English[J]. Communications of the ACM, 1995, 38(11): 39-41.

[18] WordNet Interface.

基于计数的方法的单词向量

[19] Church, Kenneth Ward, and Patrick Hanks.Word association

norms,mutual information, and lexicography[J]. Computational linguistics, 1990, 16(1): 22-29.

[20] Deerwester, Scott, et al.Indexing by latent semantic analysis[J]. Journal of the American society for information science, 1990, 41(6): 391.

[21] TruncatedSVD.

word2vec 相关

[22] Mikolov, Tomas, et al.Efficient estimation of word representations in vector space[J]. arXiv preprint arXiv:1301.3781, 2013.

[23] Mikolov, Tomas, et al.Distributed representations of words and phrases and their compositionality[J]. Advances in neural information processing systems. 2013.

[24] Baroni, Marco, Georgiana Dinu, and Germán Kruszewski. Don't count, predict! A systematic comparison of context-counting vs. context-predicting semantic vectors[J]. ACL (1), 2014.

[25] Levy, Omer, Yoav Goldberg, Ido Dagan. Improving distributional similarity with lessons learned from word embeddings[J]. Transactions of the Association for Computational Linguistics, 2015, 3: 211-225.

[26] Levy, Omer, Yoav Goldberg.Neural word embedding as implicit matrix factorization[J]. Advances in neural information processing systems, 2014.

[27] Pennington, Jeffrey, Richard Socher, Christopher D. Manning.Glove: Global Vectors for Word Representation[J]. EMNLP. Vol.14. 2014.

[28] Bengio, Yoshua, et al. A neural probabilistic language model[J]. Journal of machine learning research, 2003, 3(Feb): 1137-1155.

RNN 相关

[29] Talathi, Sachin S., Aniket Vartak. Improving performance of recurrent neural network with relu nonlinearity[J]. arXiv preprint arXiv:1511.03771, 2015.

[30] Pascanu, Razvan, Tomas Mikolov, Yoshua Bengio.On the difficulty of training recurrent neural networks[J]. International Conference on Machine Learning, 2013.

[31] Understanding LSTM Networks.

[32] Chung, Junyoung, et al.Empirical evaluation of gated recurrent neural networks on sequence modeling[J]. arXiv preprint arXiv:1412.3555, 2014.

[33] Jozefowicz, Rafal, Wojciech Zaremba, Ilya Sutskever.An empirical exploration of recurrent network architectures[J]. International Conference on Machine Learning, 2015.

基于 RNN 的语言模型

[34] Merity, Stephen, Nitish Shirish Keskar, Richard Socher.Regularizing and optimizing LSTM language models[J]. arXiv preprint arXiv:1708.02182, 2017.

[35] Zaremba, Wojciech, Ilya Sutskever, Oriol Vinyals.Recurrent neural network regularization[J]. arXiv preprint arXiv:1409.2329, 2014.

[36] Gal, Yarin, Zoubin Ghahramani.A theoretically grounded application of dropout in recurrent neural networks[J].Advances in neural information processing systems, 2016.

[37] Press, Ofir, Lior Wolf.Using the output embedding to improve language models[J].arXiv preprint arXiv:1608.05859, 2016.

[38] Inan, Hakan, Khashayar Khosravi, Richard Socher.Tying Word

Vectors and Word Classifiers: A Loss Framework for Language Modeling[J]. arXiv preprint arXiv:1611.01462, 2016.

[39] Word-level language modeling RNN.

seq2seq 相关

[40] Implementation of sequence to sequence learning for performing addition of two numbers (as strings).

[41] Sutskever, Ilya, Oriol Vinyals, Quoc V. Le.Sequence to sequence learning with neural networks[J]. Advances in neural information processing systems, 2014.

[42] Cho, Kyunghyun, et al.Learning phrase representations using RNN encoder-decoder for statistical machine translation[J]. arXiv preprint arXiv:1406.1078, 2014.

[43] Vinyals, Oriol, Quoc Le.A neural conversational model[J]. arXiv preprint arXiv:1506.05869, 2015.

[44] Zaremba, Wojciech, Ilya Sutskever.Learning to execute[J]. arXiv preprint arXiv:1410.4615, 2014.

[45] Vinyals, Oriol, et al.Show and tell: A neural image caption generator[J].Computer Vision and Pattern Recognition (CVPR), 2015 IEEE Conference on. IEEE, 2015.

[46] Karpathy, Andrej, Li Fei-Fei.Deep visual-semantic alignments for generating image descriptions[J]. Proceedings of the IEEE conference on computer vision and pattern recognition, 2015.

[47] Show and Tell: A Neural Image Caption Generator.

Attention 相关

[48] Bahdanau, Dzmitry, Kyunghyun Cho, Yoshua Bengio：Neural machine translation by jointly learning to align and translate[J].

arXiv preprint arXiv:1409.0473, 2014.

[49] Luong, Minh-Thang, Hieu Pham, Christopher D. Manning.Effective approaches to attention-based neural machine translation[J]. arXiv prelpreprint arXiv:1508.04025, 2015.

[50] Wu, Yonghui, et al. Google's neural machine translation system: Bridging the gap between human and machine translation[J]. arXiv preprint arXiv:1609.08144, 2016.

[51] A Neural Network for Machine Translation, at Production Scale.

[52] Vaswani, Ashish, et al.Attention Is All You Need[J].arXiv preprint arXiv:1706.03762, 2017.

[53] Transformer: A Novel Neural Network Architecture for Language Understanding.

[54] Gehring, Jonas, et al.Convolutional Sequence to Sequence Learning[J]. arXiv preprint arXiv:1705.03122, 2017.

带外部存储的 RNN

[55] Graves, Alex, Greg Wayne, Ivo Danihelka,Neural turing machines[J]. arXiv preprint arXiv:1410.5401, 2014.

[56] Graves, Alex, et al.Hybrid computing using a neural network with dynamic external memory[J]. 2016, Nature 538(7626): 471.

[57] Differentiable neural computers.

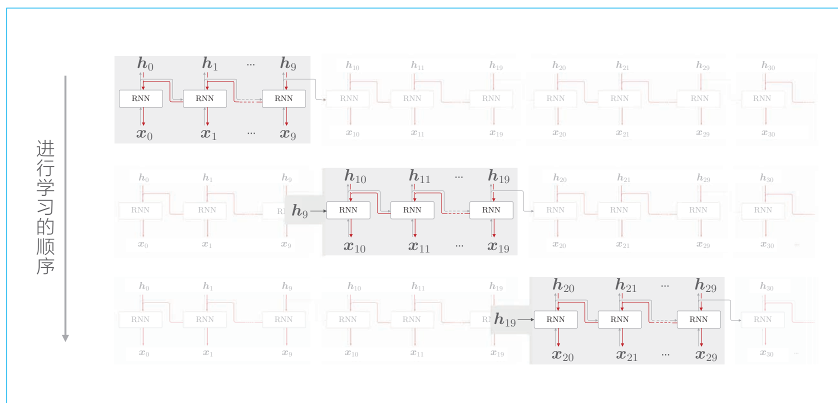


图5-14　Truncated BPTT的数据处理顺序

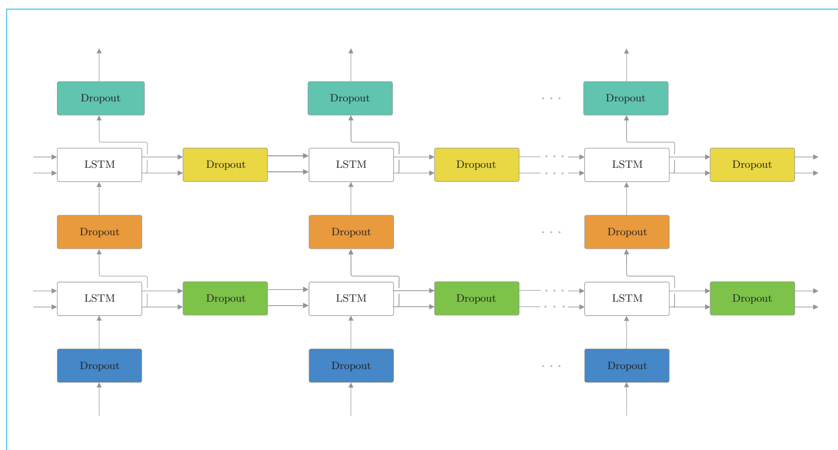


图6-34　变分Dropout的例子：具有相同图示的Dropout使用相同的mask。像这样，位于同一层的Dropout使用相同的mask，对时间方向上的Dropout也有效果

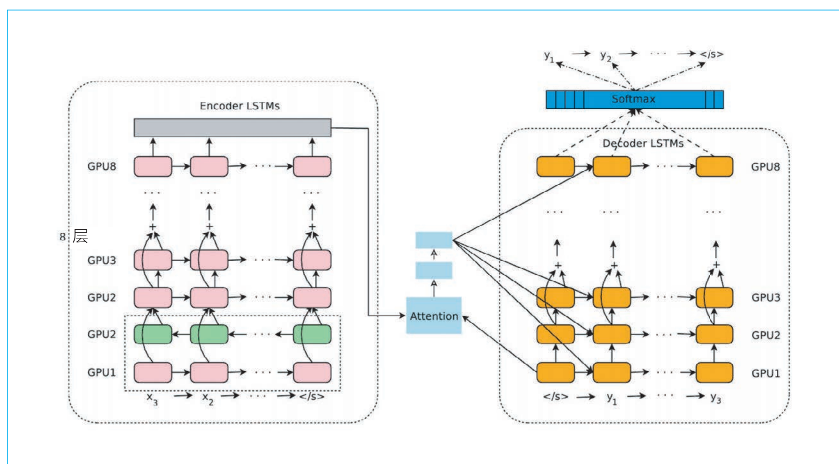


图8-35　GNMT的层结构（引自文献[50]）

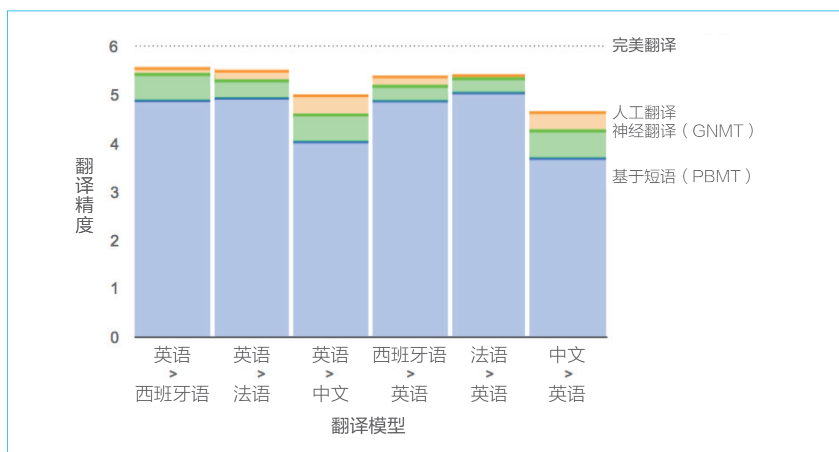


图8-36　GNMT的精度评价：纵轴是人按照0~6对翻译质量进行的评价（引自文献[51]）



微信连接



回复“深度学习”查看相关书单



微博连接

关注@图灵教育 每日分享IT好书



QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

图灵社区
iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈

O'REILLY®
オライリー・ジャパン

深度学习进阶：自然语言处理

畅销书《深度学习入门：基于Python的理论与实现》续作

带你快速直达自然语言处理领域！

本书有什么特点？

■ **简明易懂**

本书内容精炼，聚焦深度学习视角下的自然语言处理，延续前作的行文风格，采用通俗的语言和大量直观的示意图详细讲解，帮助读者加深对深度学习技术的理解，轻松入门自然语言处理。

■ **侧重原理**

不依赖外部库，使用Python 3从零开始创建深度学习程序，通过亲自创建程序并运行，读者可透彻掌握word2vec、RNN、LSTM、GRU、seq2seq和Attention等技术背后的运行原理。

■ **学习曲线平缓**

按照“文字介绍→代码实现→分析结果→发现问题→进行改善”的流程，逐步深入，读者只需具备基础的神经网络和Python知识，即可轻松读懂。

图灵社区：iTuring.cn

分类建议　计算机／人工智能



扫码获取
随书代码资料

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Japan, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-54764-4



定价：99.00元